



US 20030018926A1

(19) **United States**(12) **Patent Application Publication**  
Chen(10) Pub. No.: **US 2003/0018926 A1**(43) Pub. Date: **Jan. 23, 2003**(54) **TRANSITION TO SWITCH NODE ADAPTER  
DIAGNOSTICS USING ADAPTER DEVICE  
DRIVER**

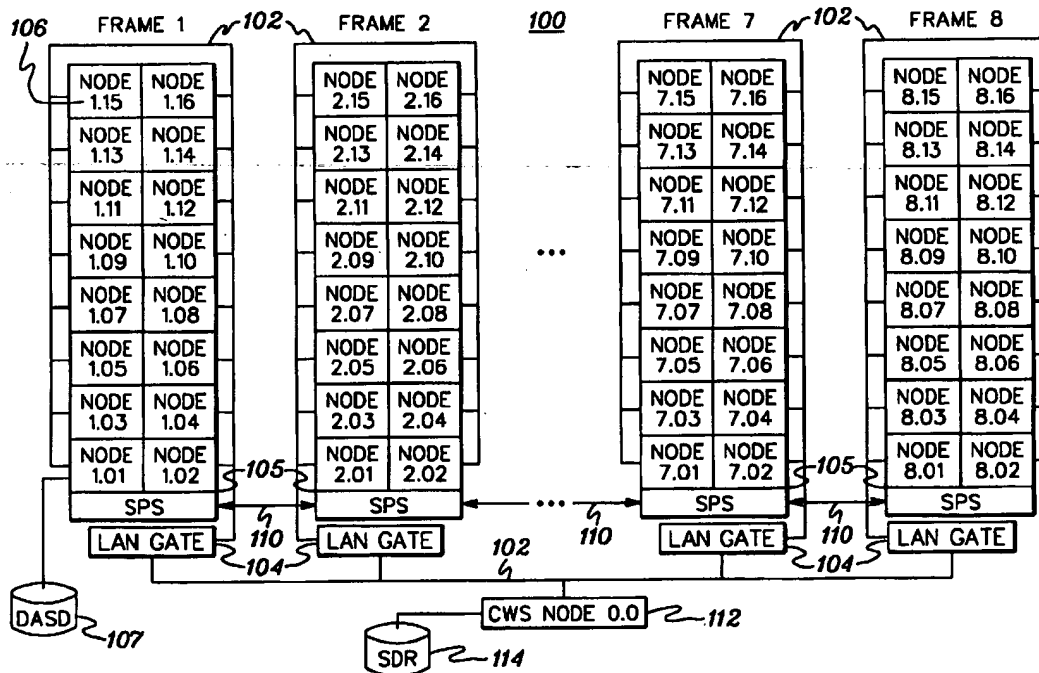
(52) U.S. Cl. .... 714/4

(75) Inventor: **Wen C. Chen, Rhinebeck, NY (US)**(57) **ABSTRACT**

Correspondence Address:

**Kevin P. Radigan, Esq.****HESLIN ROTHENBERG FARLEY & MESITI  
P.C.****5 Columbia Circle****Albany, NY 12203 (US)**(73) Assignee: **International Business Machines Cor-  
poration, Armonk, NY**(21) Appl. No.: **09/907,118**(22) Filed: **Jul. 17, 2001****Publication Classification**(51) Int. Cl.<sup>7</sup> ..... **H02H 3/05**

Smooth release of resources on a switch node adapter to a diagnostics program is provided without requiring rebooting of the system. The release technique includes: setting a first flag at a device driver associated with the adapter to block new open system calls from opening the device driver; broadcasting an adapter down event to instruct internet protocol (IP), fault service daemon (FSD) and application program interface (API) components of the node to release resources on the adapter; and setting a second flag at the device driver to block input/output control calls other than from the diagnostics program. When the device driver supports multiple adapters, only the adapter undergoing diagnostics is suspended, the remaining adapters remain up to components of the switch node. Upon completion of diagnostics, components remaining in open state are restarted on their communication windows at the adapter.



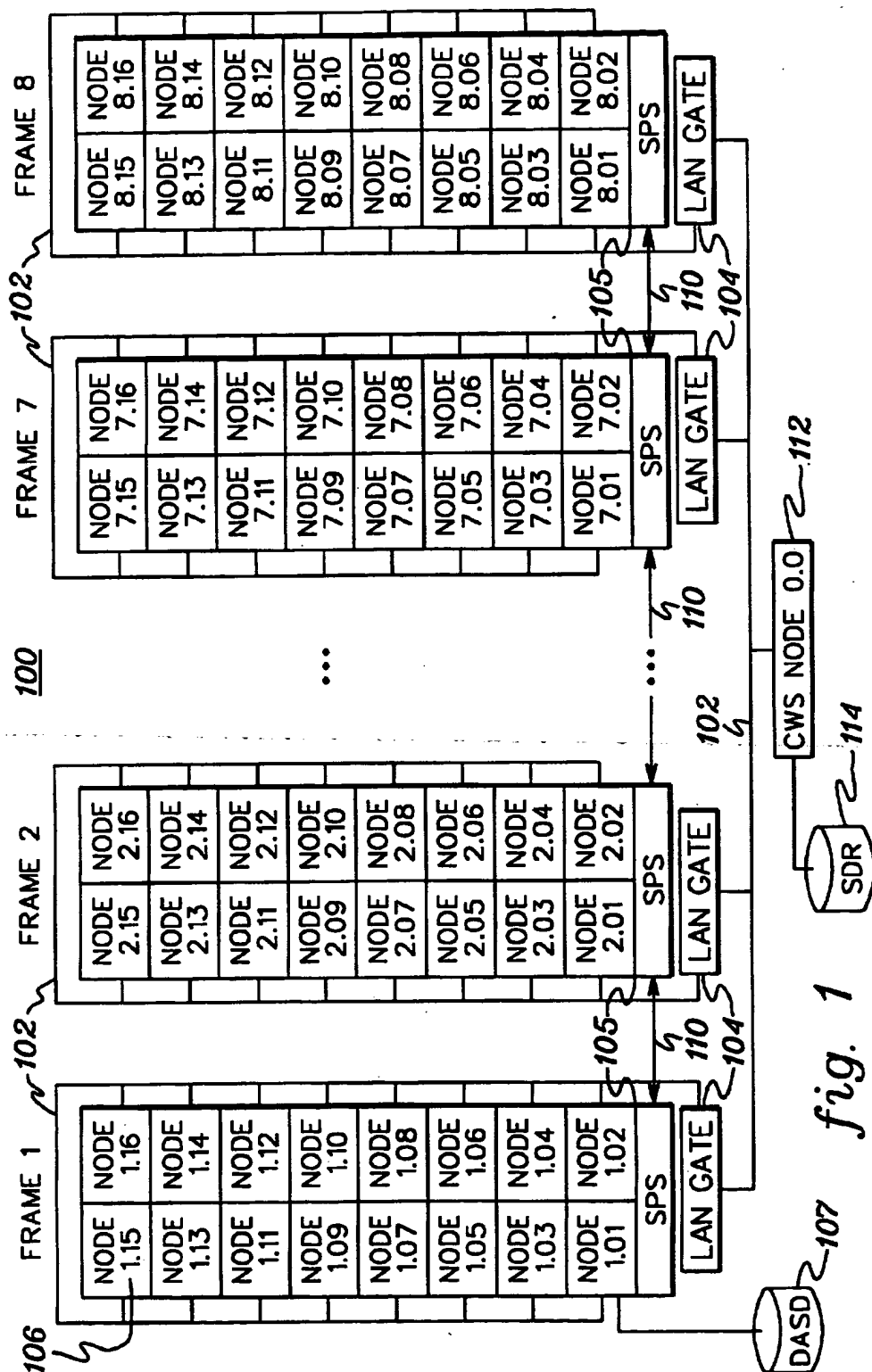


fig. 1

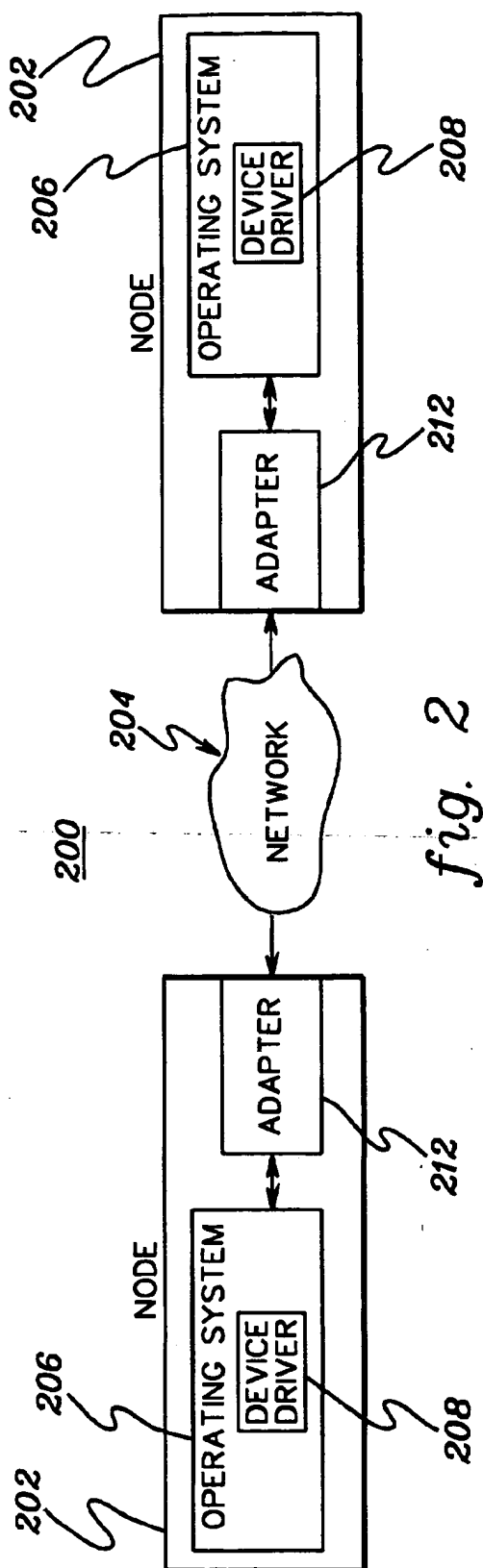


fig. 2

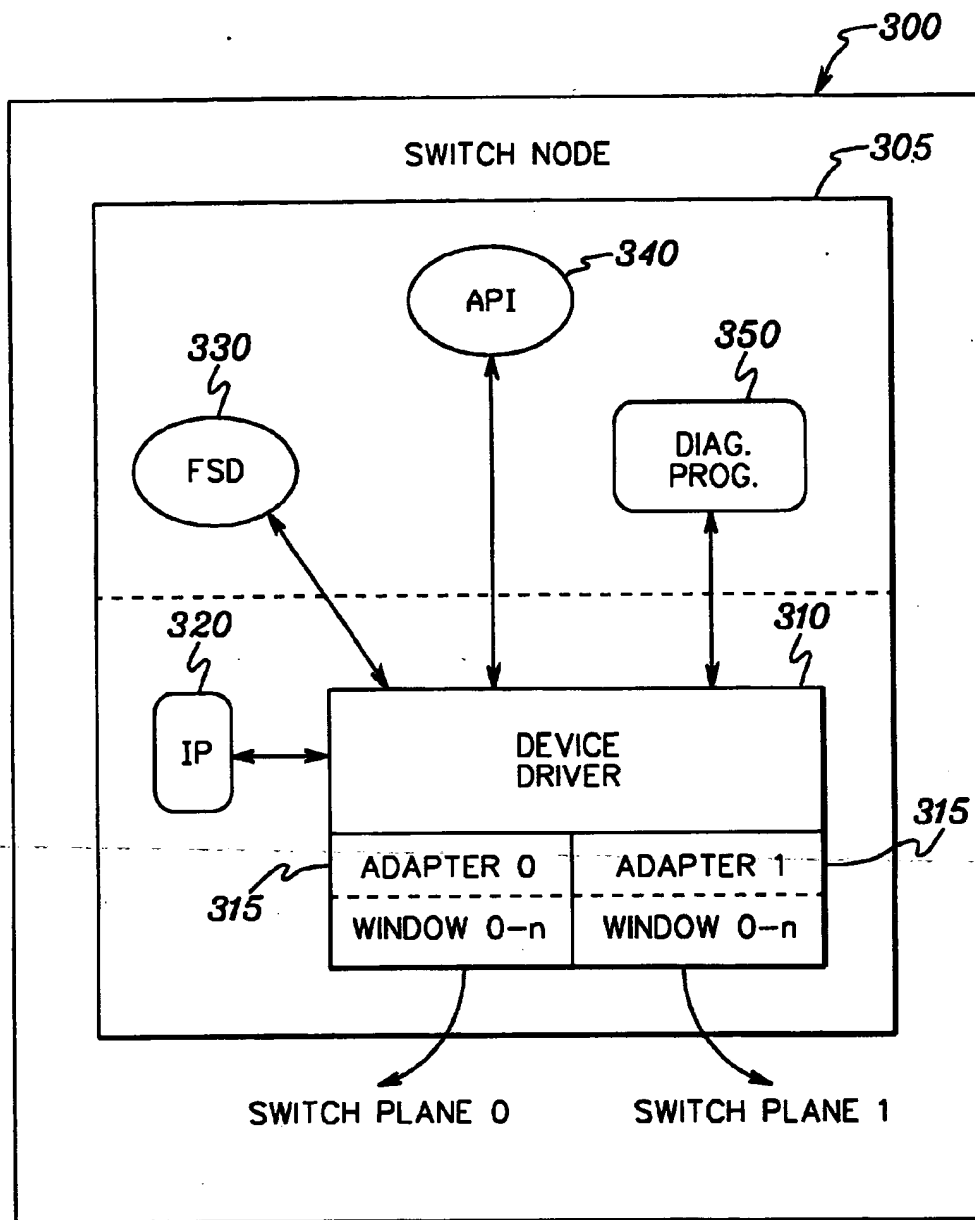
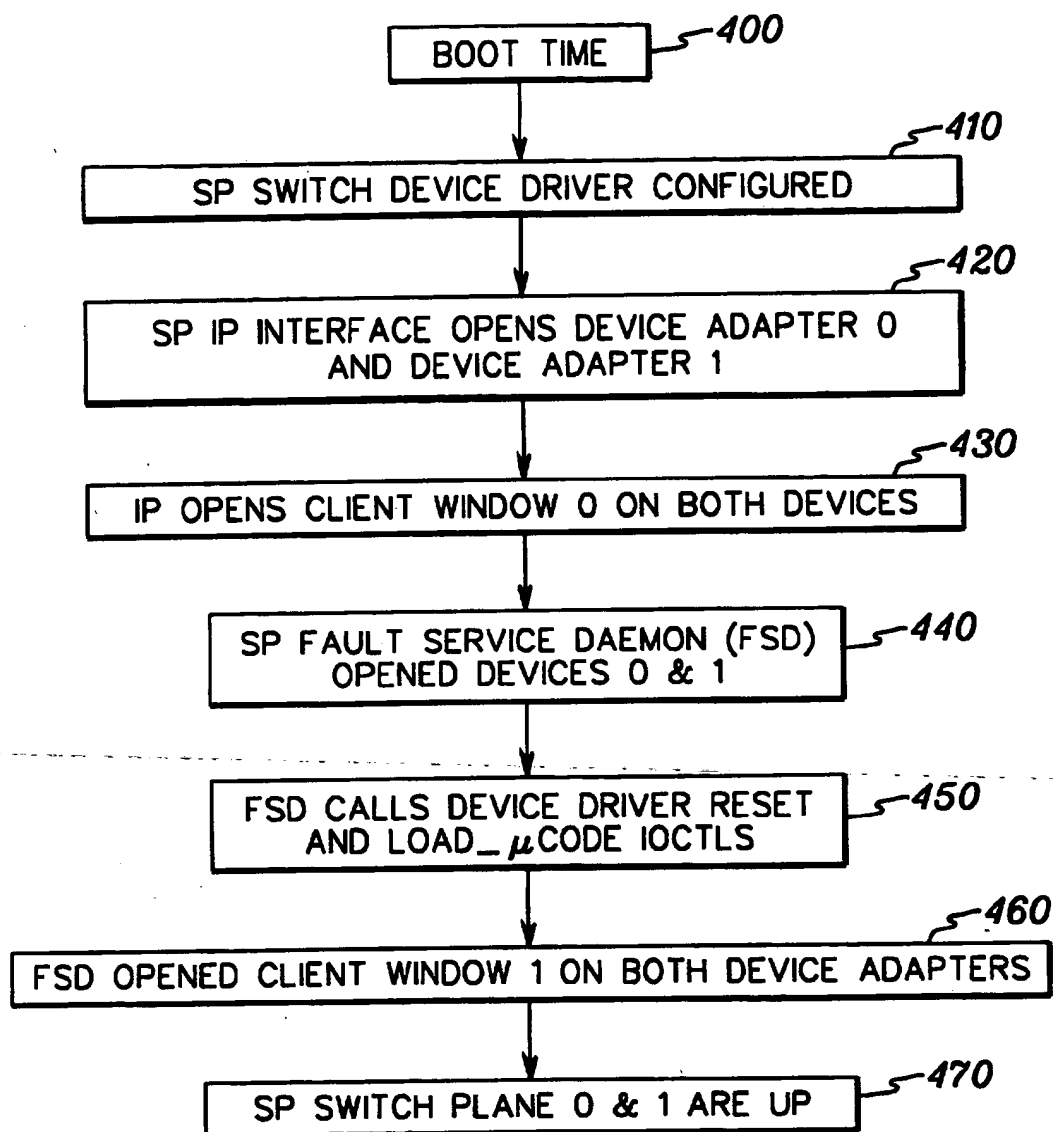
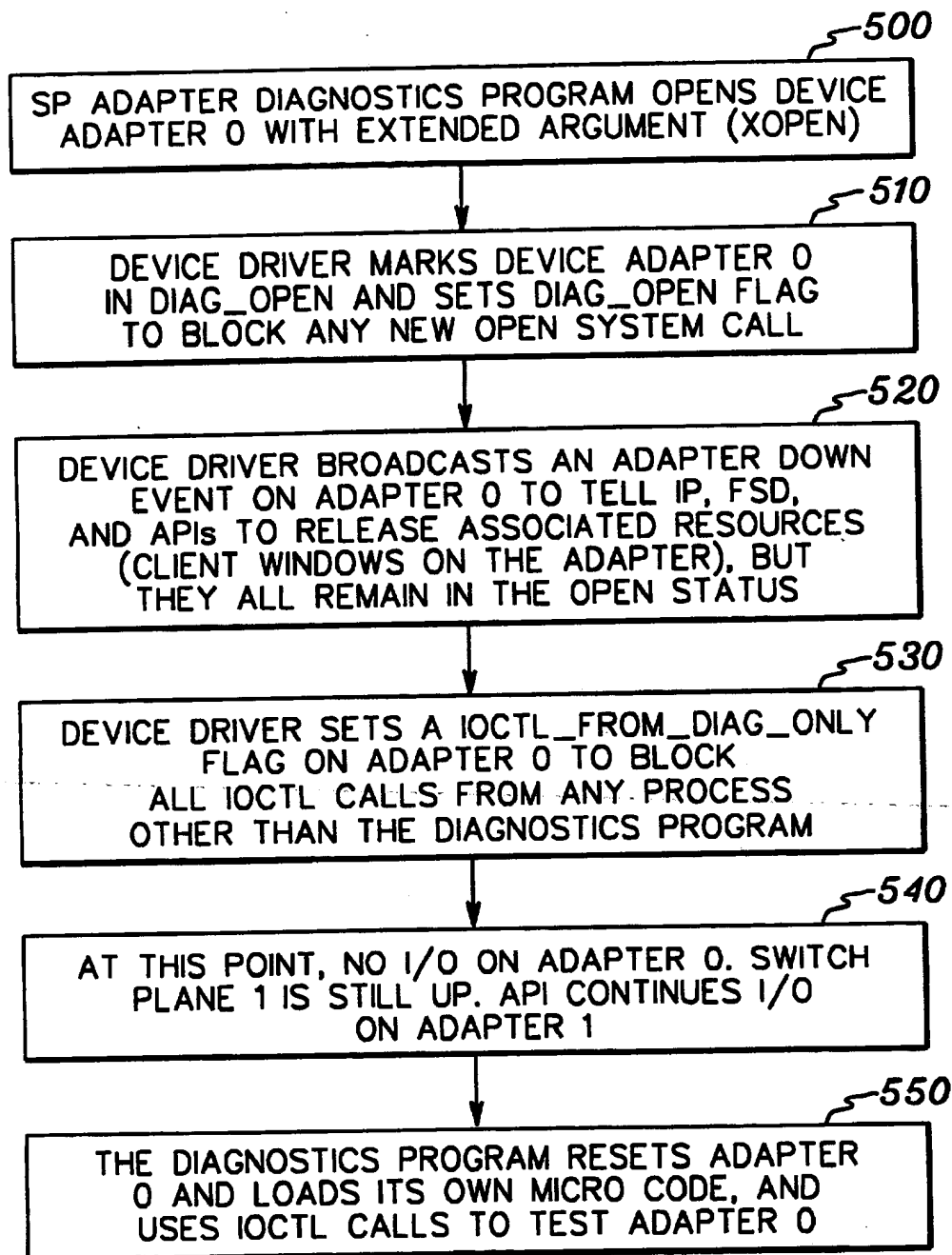
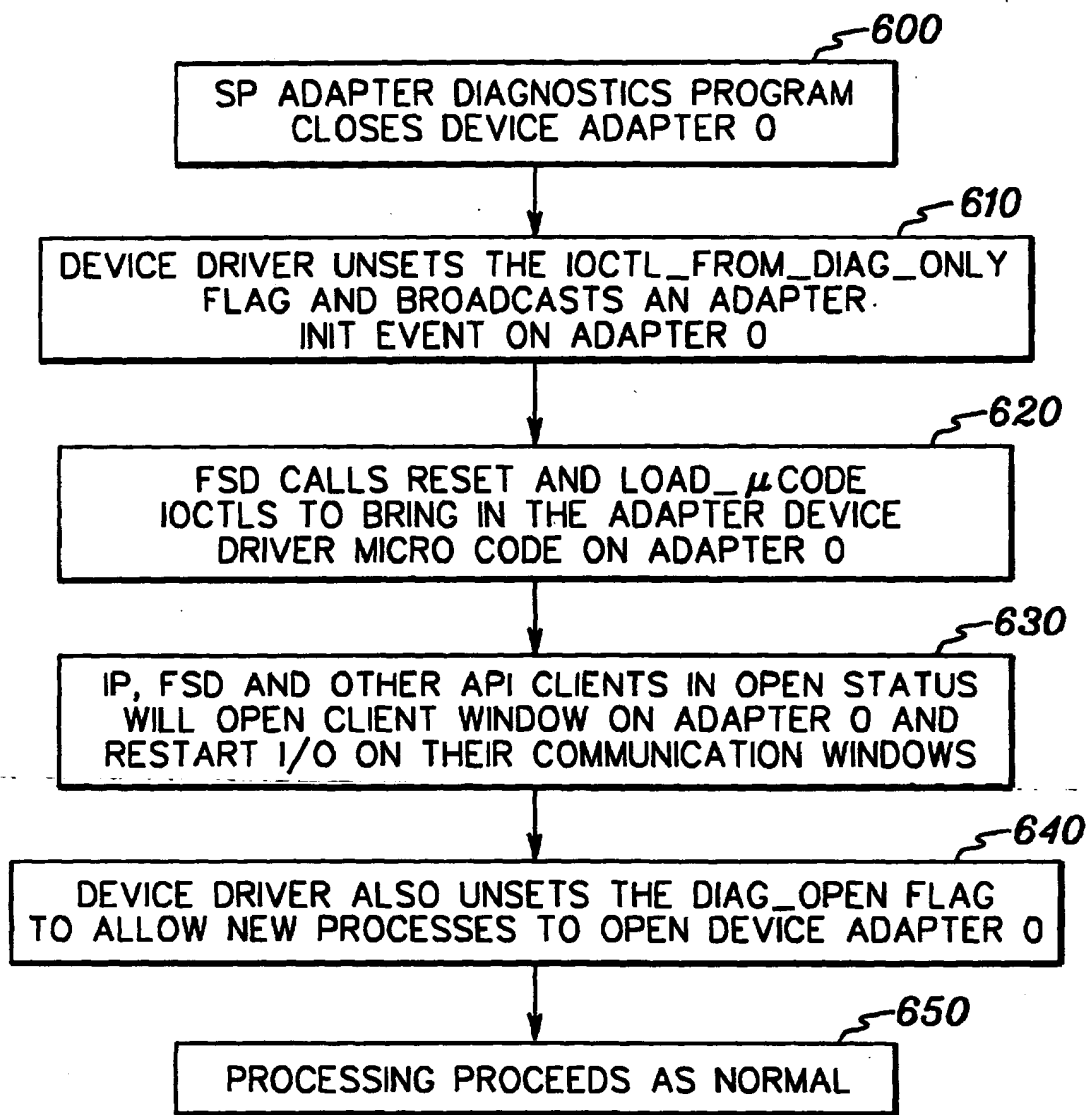


fig. 3



*fig. 4*

*fig. 5*

*fig. 6*

## TRANSITION TO SWITCH NODE ADAPTER DIAGNOSTICS USING ADAPTER DEVICE DRIVER

### TECHNICAL FIELD

[0001] This invention relates in general to a distributed computer system having a plurality of data processing nodes and one or more switch units which establish links between the plurality of processing nodes, and more particularly, to a testing technique wherein resources on a switch node adapter are smoothly released to a diagnostics program without requiring rebooting of the system.

### BACKGROUND OF THE INVENTION

[0002] In certain parallel processing systems containing a plurality of data processing nodes, one or more switching units are provided for transferring data between the data processing nodes. A switch unit receives a connection request from one of a plurality of nodes and establishes a link between the data processing node which sends the connection request, and another of the plurality of data processing nodes, which is requested by the connection request. As used herein, a "switch node" means any node of the plurality of data processing nodes which is capable of connecting to such a switch unit. A switch node includes a device driver and at least one adapter.

[0003] Conventionally, during diagnostics testing of a switch node, a diagnostics program loads special microcode to test the adapter hardware. This process typically requires a device driver of the node to stop all send and receive requests and input/output controls issued other than for the diagnostics program. Open windows on the switch adapter must be closed to allow the diagnostics program to utilize and verify them. After performing the diagnostics, the switch node must then be rebooted to resume normal operations.

[0004] In view of the above, commercial advantage would be obtained in the art with a more seamless technique for releasing switch node resources to a diagnostics program from current clients and applications, particularly if without requiring subsequent rebooting of the system.

### SUMMARY OF THE INVENTION

[0005] The shortcomings of the prior art are overcome and additional advantages are provided through the provision of a method of releasing a switch node adapter to a diagnostics program. The method includes: setting a first flag at a device driver associated with the adapter to block any new open system call; broadcasting an adapter down event to instruct components of the switch node to release resources on the adapter without closing those resources; and setting a second flag at the device driver to block input/output (I/O) control calls other than from the diagnostics program. If the device driver supports multiple adapters, then the other adapter(s) remain up and support communication I/O during diagnostic testing of the one adapter. Upon completion of diagnostic testing of the adapter, all blocked clients and applications of that adapter are resumed if they are still open and awaiting adapter resources.

[0006] Various method enhancements, as well as systems and computer program products corresponding to the above-summarized method and its enhancements are also described and claimed herein.

[0007] To restate, provided herein in one aspect is a technique for smoothly releasing resources on a switch node adapter to a diagnostics program. Advantageously, the technique will not result in the device driver being unconfigured, nor require rebooting of the system after testing. Thus, the process set forth will not affect the system nor applications on the system. Additionally, if the device driver supports multiple adapters, then the switch adapter diagnostics can be testing one adapter, while another adapter remains functioning.

[0008] Additional features and advantages are realized through the techniques of the present invention. Other embodiments and aspects of the invention are described in detail herein and are considered a part of the claimed invention.

### BRIEF DESCRIPTION OF THE DRAWINGS

[0009] The subject matter which is regarded as the invention is particularly pointed out and distinctly claimed in the claims at the conclusion of the specification. The foregoing and other objects, features, and advantages of the invention are apparent from the following detailed description taken in conjunction with the accompanying drawings in which:

[0010] FIG. 1 depicts one example of a communications environment incorporating and using one or more aspects of the present invention;

[0011] FIG. 2 depicts further details regarding a plurality of nodes of the communications environment of FIG. 1 incorporating and using one or more aspects of the present invention;

[0012] FIG. 3 depicts one embodiment of a switch node having a device driver and multiple switch adapters to undergo diagnostic testing in accordance with an aspect of the present invention;

[0013] FIG. 4 is a flowchart of one embodiment of processing implemented by the switch node of FIG. 3 at boot time;

[0014] FIG. 5 is a flowchart of one embodiment of diagnostics start up processing implemented by the switch node of FIG. 3 in accordance with an aspect of the present invention; and

[0015] FIG. 6 is a flowchart of one embodiment of processing implemented by the switch node of FIG. 3 upon completion of diagnostics testing of one or more switch adapters.

### BEST MODE FOR CARRYING OUT THE INVENTION

[0016] FIG. 1 is a schematic diagram of a distributed computer system 100 incorporating and using aspects of the present invention. As one example, the distributed computer system 100 may be an IBM RISC System/6000 Scalable POWERparallel Systems (SP) distributed computer system available from International Business Machines Corporation of Armonk, N.Y. The embodiment disclosed in FIG. 1 is an SP computing environment having a total of 8 frames, with each frame having up to 16 processing nodes, for a total of 128 nodes. All of the nodes 106 are joined by a local area network (LAN) 102. Each processing node 106 is a computer itself, and may (for instance) be a RISC System/6000



computer running AIX, a UNIX based operating system well-known by those skilled in the art.

[0017] All of the nodes in a frame of the SP computer 100 are included in a LAN segment which is joined by the other LAN segments through LAN gates 104. As examples, each LAN gate 104 includes either a RISC/6000 computer, any computer network connected to the LAN or a network router. However, these are only examples. It will be apparent to those skilled in the art that there are other types of LAN gates and that other mechanisms can be used to couple the frames to one another.

[0018] Also connected to the LAN 102, is a control workstation (CWS) 112 which controls operation of the SP computer 100. The control workstation has a direct access storage device (DASD) 114 referred to as the System Shared Disk on which is stored the system data repository (SDR) files. The SDR files include such information as a list of nodes that are in the system and their configuration, and includes resources and object definitions for each of the nodes 106. Each node 106 also includes a DASD device 107 for storing data processed by the SP computer 100.

[0019] In one embodiment, the nodes in each frame are also connected to an IBM Scalable POWERparallel switch (SPS) 105. Each of the SPS switches 105 of each frame is connected to neighboring SPS switches 105 of other frames by a bus 110.

[0020] As is well understood in the art, the CWS node 112 sends system data and control signals to the frames of the SP computer 100 by means of the LAN 102, while messages and data may be sent from one to another of the nodes 106 by means of the high performance switches 105.

[0021] The distributed computing environment of FIG. 1 is only one example. It is possible to have more or less than 8 frames, or more or less than 16 nodes per frame. Further, the processing nodes do not have to be RISC/6000 computers running AIX. Some or all of the processing nodes can include different types of computers and/or different operating systems. Further, aspects of the invention are usable with other types of communications environments. All of these variations are considered a part of the claimed invention.

[0022] FIG. 2 depicts a more detailed embodiment of several nodes of a computing environment such as depicted in FIG. 1 incorporating and using aspects of the present invention. Environment 200 includes, for instance, a plurality of nodes 202 coupled to one another via one or more networks 204. Although two nodes and one network are illustrated, it is well known that a computing environment can include many nodes and networks such as described above in connection with FIG. 1. Further, a particular node can be coupled to one or more other nodes via one network type, and further coupled to one or more other nodes via another network type, etc.

[0023] A node 202 includes an operating system 206, such as AIX. Operating system 206 includes, for instance, at least one communication device driver 208. The device driver is based on the type of adapter in the environment, such as SP switch-2, and it may manage one or more adapters 212.

[0024] Each network adapter is used to couple its associated node to a particular network, which corresponds to that

adapter type. Network 204 includes any IP capable network, such as the Ethernet, Token Ring, FDDI, ATM, SP Switch, Fiber Channel, etc.

[0025] More particular to the present invention, a goal of the processing technique presented herein is to smoothly release the SP switch resources to the diagnostics program from current clients and applications, without subsequently requiring rebooting of the system and device driver configuration. The concepts presented herein can be implemented in software as additional intelligent control at the device driver. The adapter device driver blocks (i.e., makes dormant) the current clients and applications until the diagnostics program is complete. After that point, the device driver resumes all the blocked clients and applications if they are still open and awaiting resources. This depends on how long the applications will keep retrying, and whether the client can stay in an open state and wait for a start event. The adapter and device driver continue to function at all times without reconfiguration. If there are two adapters on the device driver, such as the case with an SP switch-2 double/double configuration, network input/output may proceed on the second adapter while the first undergoes diagnostic testing.

[0026] FIG. 3 depicts one embodiment of a switch node incorporating and using aspects of the present invention. As one detailed example, switch node 300 is discussed herein to comprise an SP node, however, those skilled in the art will recognize that the concepts disclosed are applicable to other types of switch nodes, and that the claims presented should not be read as limited to an SP node or an SP switch adapter per se.

[0027] As noted above, an SP switch adapter diagnostics program loads special microcode to test adapter hardware. To accomplish this, the switch device driver must stop all send and receive requests and ioctls issued other than those from the diagnostics program. For example, no more IP packets could be sent nor received on the SP switch adapter until diagnostics testing is complete. Windows (i.e., partitions) on the SP switch adapter must be closed to allow the diagnostics program to utilize and verify them.

[0028] The problem with this approach is that the SP switch control program RC.SWITCH is always run at the boot INIT time. IP and SP switch fault service daemon (FSD) are configured and windows are opened and started. A system administrator must then manually create a dummy RC.SWITCH file not to configure IP and FSD, but to reboot the SP node to close all the windows (partitions) on the SP switch adapter to allow for diagnostics testing. Furthermore, if there are two SP adapters on a given node, then both must be closed, i.e., neither can be used for communications when the diagnostics program is testing one of the adapters.

[0029] As noted, presented herein is an online SP switch adapter diagnostic support technique which operates from the adapter's device driver and which will not unconfigure the device and reboot the system to accomplish diagnostic testing of the adapter. Further, if the SP switch adapter diagnostics is testing one adapter in a multiadapter environment, the other adapter(s) is still functional.

[0030] Continuing with FIG. 3, one embodiment of a switch node 300 is shown at which operating system 305 resides. The operating system 305 includes various system

components, including a device driver 310 and two adapters 315. Device driver 310 and adapters 315 allow internet protocol (IP) 320, fault service daemon (FSD) 330 and application program interfaces (APIs) 340 components to communicate via a switching network (not shown) with other nodes in a distributed processing system such as described above. Each adapter 315 (i.e., adapter 0, adapter 1) allows components 320, 330 & 340 to access a different switch plane (e.g., switch plane 0, switch plane 1). Components 320, 330 & 340 open windows at the different adapters which allow communication via the switching network with other nodes in the system. These windows in FIG. 3 are labeled 0-n, where the number of windows n is system dependent. One example of a switch node 300 would be an SP node such as described above in connection with FIGS. 1 & 2. Further, the device driver 310 could comprise an SP\_Switch2 device driver marketed by International Business Machines Corporation of Armonk, N.Y. Switch adapter 315 testing is accomplished by use of a diagnostics program 350.

[0031] In accordance with an aspect of the present invention, intelligence is added to device driver 310 to accomplish a smooth release of control of adapter 315 to the diagnostics program. This includes setting a first flag at the device driver associated with the adapter to block any new open system call, and broadcasting an adapter down event from the device driver to other components of the node to instruct those components to release resources on the adapter. These resources would comprise windows as shown in FIG. 3. A second flag is then set by the device driver to block input/output control calls other than from the diagnostics program itself.

[0032] FIG. 4 depicts a flowchart of one embodiment of processing performed by switch node 300 at operating system boot time 400. In this example, the SP switch device driver, such as an SP switch\_2 device driver is configured 410, after which the SP IP interface opens adapter devices 0 & 1 420. The IP opens a client window 0 on both device adapters 430, and an SP fault service daemon (FSD) also opens device adapter 0 & 1 440. The FSD component then calls the device driver to reset and load input/output control microcode (load  $\mu$ code ioctls) 450. The FSD opens client window 1 on both adapter devices 460, after which two SP switch planes 0 & 1 are up and functional 470. Each component 320, 330 & 340 (see FIG. 3) uses at least one port or window in adapter 0 and in adapter 1.

[0033] After boot time, the API component opens the device driver, and opens client windows on either or both device adapters based on the client type in the user space program. Alternatively, the API component can accomplish input/output via the SP IP interface.

[0034] One embodiment of release of resources and diagnostic testing in accordance with the principles of the present invention is depicted in FIG. 5. The SP adapter diagnostics program first opens a device adapter 0 with an extended argument 500. More particularly, the SP switch diagnostics program opens the switch adapter device with, for example, a UNIX "OPENX" system call to pass an "EXT" parameter to tell the adapter device that a diagnostics mode is to be initiated. The device driver then marks (for example) adapter device 0 with diag\_open and sets a diag\_open flag to block any new open system calls to open that

adapter 510. The device driver also broadcasts an adapter down event to ask all users, including the IP component to release their resources on the adapter, i.e., windows or partitions on the adapter 520. (In accordance with this invention, all released resources can remain in open status.)

[0035] This is accomplished through a list of input/output control (ioctls) calls from the users of the windows. An adapter down request is also put on the fault service daemon's request queue. The FSD processes this request to close the service window and terminate the adapter thread. After this, the SP switch device driver sets an ioctl\_from\_diag\_only flag to block any ioctl calls issued from any process other than the diagnostics program based on process id 530. This completes the device open routine called by the diagnostics program. From this time on all ioctl calls and send/receive requests will only be accepted from the diagnostics program. The diag\_open and ioctl\_from\_diag\_only flags are set and associated with the device adapter to be tested. When the diagnostics program is testing one adapter in a multiadapter environment, the other adapter(s) is still capable of functioning. For example, the API component continues to receive and send input/output on device adapter 1, while device adapter 0 is undergoing diagnostics testing 540. Next, the diagnostics program resets the device adapter to be tested (e.g., adapter 0), loads its own microcode to the adapter, and uses ioctl calls to test adapter 0 550.

[0036] Once testing is completed, the diagnostic program calls the device close function to close device adapter 0 600 (FIG. 6). The device driver then unsets the ioctl\_from\_diag\_only flag and broadcasts an adapter "up" event to let the FSD, IP and other open clients reopen their client windows on the adapter 610. The FSD then calls the reset and load  $\mu$ code ioctls to bring in the adapter device driver microcode 620. IP, FSD and other API clients in open status will open client windows on the adapter and restart I/O on their communication windows 630. The device driver also unsets the diag\_open flag to allow new processes to open device adapter 0 640. Thus, the adapter is back to normal processing 650.

[0037] The present invention can be included in an article of manufacture (e.g., one or more computer program products) having, for instance, computer usable media. The media has embodied therein, for instance, computer readable program code means for providing and facilitating the capabilities of the present invention. The article of manufacture can be included as a part of a computer system or sold separately.

[0038] Additionally, at least one program storage device readable by a machine, tangibly embodying at least one program of instructions executable by the machine to perform the capabilities of the present invention can be provided.

[0039] The flow diagrams depicted herein are just examples. There may be many variations to these diagrams or the steps (or operations) described therein without departing from the spirit of the invention. For instance, the steps may be performed in a differing order, or steps may be added, deleted or modified. All of these variations are considered a part of the claimed invention.

[0040] Although preferred embodiments have been depicted and described in detail herein, it will be apparent to

those skilled in the relevant art that various modifications, additions, substitutions and the like can be made without departing from the spirit of the invention and these are therefore considered to be within the scope of the invention as defined in the following claims.

What is claimed is:

1. In a switch node, a method of releasing control of an adapter to a diagnostics program, the method comprising:

setting a first flag at a device driver associated with the adapter to block any new open system call;

broadcasting an adapter down event to instruct components of the switch node to release resources on the adapter; and

setting a second flag at the device driver to block input/output control calls other than from the diagnostics program.

2. The method of claim 1, wherein the components of the switch node release resources on the adapter without closing those resources.

3. The method of claim 2, wherein the components of the switch node include one or more of an internet protocol (IP) component, a fault service daemon (FSD) component, and an application program interface (API) component.

4. The method of claim 3, wherein the resources to be released on the adapter comprise client windows on the adapter.

5. The method of claim 1, further comprising initiating the diagnostics program and opening the device driver to the diagnostics program.

6. The method of claim 1, wherein said broadcasting comprises broadcasting from the device driver the adapter down event to instruct components to release resources on the adapter.

7. The method of claim 1, wherein the first flag comprises a diag\_open flag, and the second flag comprises an ioctl\_from\_diag\_only flag.

8. The method of claim 1, wherein the adapter comprises a first adapter and the device driver further includes a second adapter, and wherein the method comprises releasing the first adapter to the diagnostics program, while the second adapter remains up and running to components of the switch node.

9. The method of claim 1, further in combination with a method of diagnostic testing of the adapter, said combination including the diagnostic program resetting the adapter, loading diagnostic microcode onto the adapter, and using ioctl calls to test the adapter.

10. The combination method of claim 9, further comprising reinitiating the adapter when diagnostics is complete, said reinitiating comprising:

closing the diagnostics program on the device driver;

unsetting the second flag and broadcasting an adapter initiate event;

bringing up default device driver microcode;

opening client windows on the adapter for any components remaining in open status, and restarting input/output on their communication windows; and

unsetting the first flag to allow new processes to open the device driver.

11. A system of initiating diagnostics of a switch node adapter, said system comprising:

means for setting a first flag at a device driver associated with the adapter to block any new open system call;

means for broadcasting an adapter down event to instruct components of the switch node to release resources on the adapter; and

means for setting a second flag at the device driver to block input/output control (ioctl) calls other than from a diagnostics program.

12. The system of claim 11, wherein the components of the switch node release resources on the adapter, without closing those resources, responsive to said means for broadcasting.

13. The system of claim 12, wherein the components of the switch node include one or more of an internet protocol (IP) component, a fault service daemon (FSD) component, and an application program interface (API) component.

14. The system of claim 13, wherein the resources to be released on the adapter comprise client windows on the adapter.

15. The system of claim 11, further comprising means for initiating the diagnostics program and for opening the device driver to the diagnostics program.

16. The system of claim 11, wherein said means for broadcasting comprises means for broadcasting from the device driver the adapter down event to instruct the components to release resources on the adapter.

17. The system of claim 11, wherein the first flag comprises a diag\_open flag, and the second flag comprises an ioctl\_from\_diag\_only flag.

18. The system of claim 11, wherein the adapter comprises a first adapter and the device driver further includes a second adapter, and wherein the system further comprises means for releasing the first adapter to the diagnostics program, while the second adapter remains up and running to components of the switch node.

19. The system of claim 11, further in combination with a system for diagnostic testing of the adapter, said combination system including means for the diagnostic program to reset the adapter, to load diagnostic microcode onto the adapter, and to use ioctl calls to test the adapter.

20. The combination system of claim 19, further comprising means for reinitiating the adapter when diagnostics is complete, said means for reinitiating comprising:

means for closing the diagnostics program on the device driver;

means for unsetting the second flag and broadcasting an adapter initiate event;

means for bringing up default device driver microcode;

means for opening client windows on the adapter for any components remaining in open status, and for restarting input/output on their communication windows; and

means for unsetting the first flag to allow new processes to open the device driver.

21. A system for facilitating diagnostics testing of a switch node adapter, said system comprising:

a device driver having an adapter to undergo diagnostics testing, wherein upon initiation of diagnostics said device driver is adapted to:

set a first flag associated with the adapter to block any new open system call;

broadcast an adapter down event to instruct components of the switch node to release resources on the adapter; and

set a second flag on the device driver to block input/output control calls other than from a diagnostics routine, thereby releasing control of the adapter to the diagnostics routine for testing.

22. At least one program storage device readable by a machine, tangibly embodying at least one program of instructions executable by the machine to perform a method of releasing control of a switch node adapter to a diagnostics program, comprising:

setting a first flag at a device driver associated with the adapter to block any new open system call;

broadcasting an adapter down event to instruct components of the switch node to release resources on the adapter; and

setting a second flag at the device driver to block input/output control calls other than from the diagnostics program.

23. The at least one program storage device of claim 22, wherein the components of the switch node release resources on the adapter without closing those resources.

24. The at least one program storage device of claim 23, wherein the components of the switch node include one or more of an internet protocol (IP) component, a fault service daemon (FSD) component, and an application program interface (API) component.

25. The at least one program storage device of claim 24, wherein the resources to be released on the adapter comprise client windows on the adapter.

26. The at least one program storage device of claim 22, further comprising initiating the diagnostics program and opening the device driver to the diagnostics program.

27. The at least one program storage device of claim 22, wherein said broadcasting comprises broadcasting from the device driver the adapter down event to instruct components to release resources on the adapter.

28. The at least one program storage device of claim 22, wherein the first flag comprises a diag\_open flag and the second flag comprises an ioctl\_from\_diag\_only flag.

29. The at least one program storage device of claim 22, wherein the adapter comprises a first adapter and the device driver further includes a second adapter, and wherein the method comprises releasing the first adapter to the diagnostics program, while the second adapter remains up and running to components of the switch node.

30. The at least one program storage device of claim 22, wherein said method is further in combination with a method of diagnostic testing of the adapter, said combination method including employing the diagnostic program to reset the adapter, to load diagnostic microcode onto the adapter, and use ioctl calls to test the adapter.

31. The at least one program storage device of claim 30, further comprising reinitiating the adapter when diagnostics is complete, said reinitiating comprising:

closing the diagnostics program on the device driver;

unsetting the second flag and broadcasting an adapter initiate event;

bringing up default device driver microcode;

opening client windows on the adapter for any components remaining in open status, and restarting input/output on their communication windows; and

unsetting the first flag to allow new processes to open the device driver.

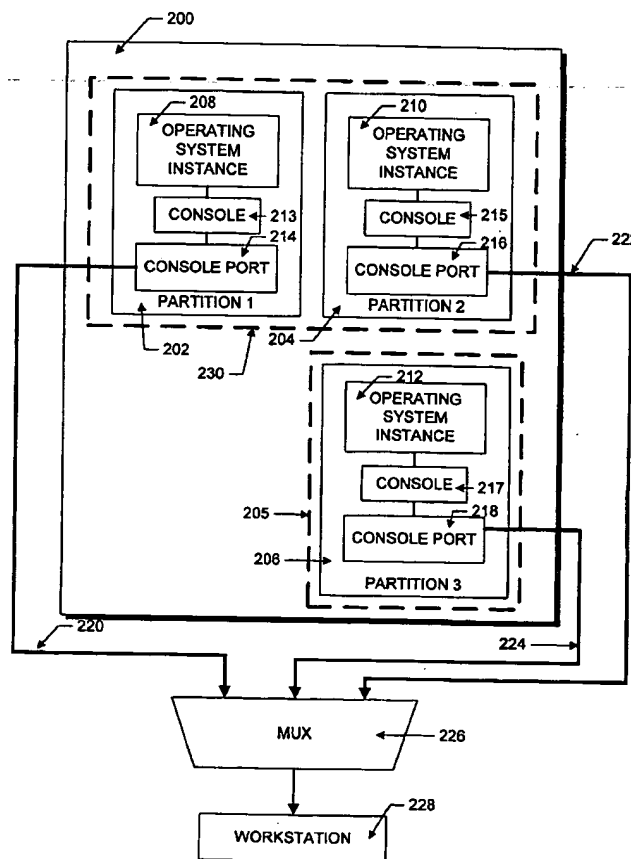
\* \* \* \* \*

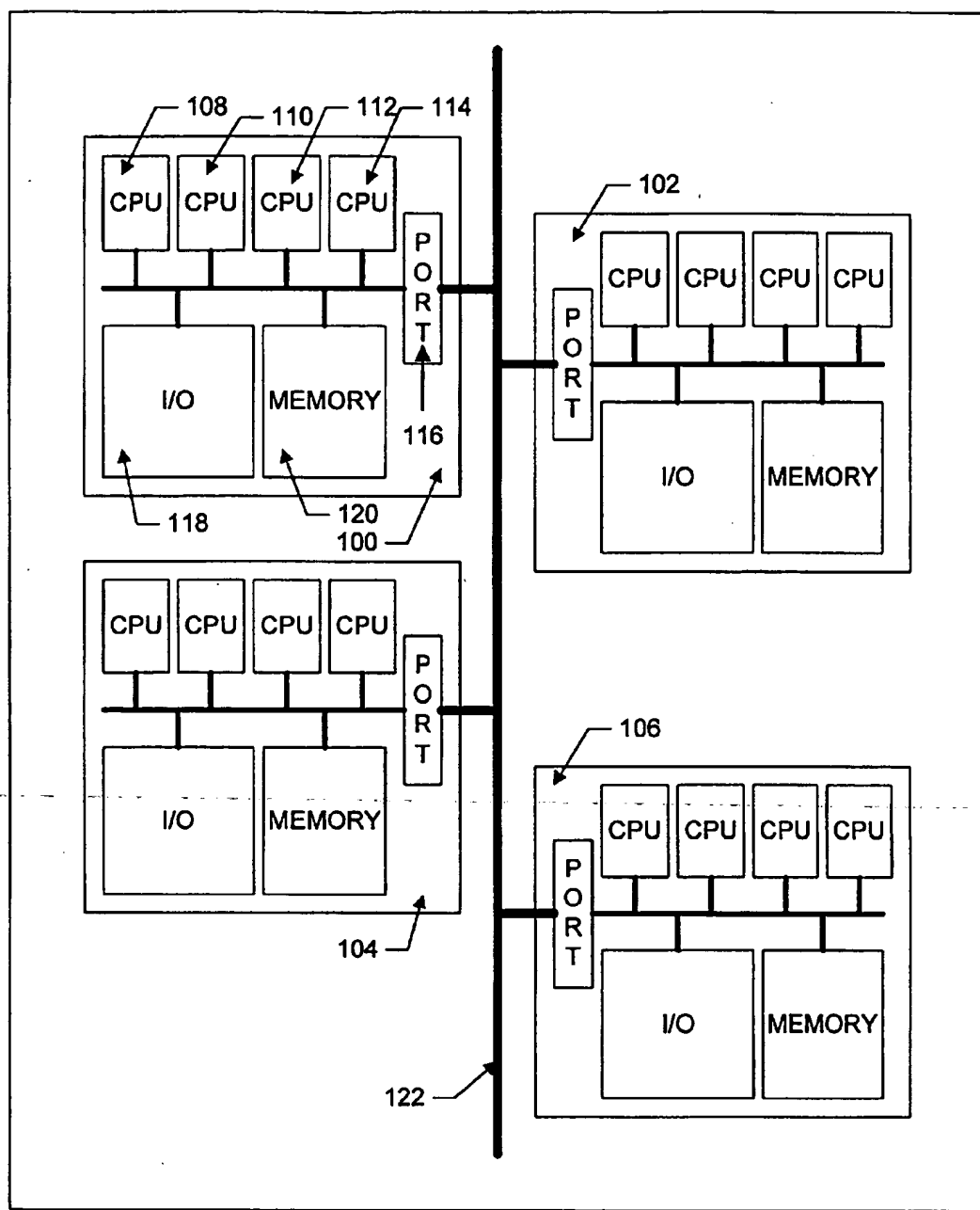


US 20020010844A1

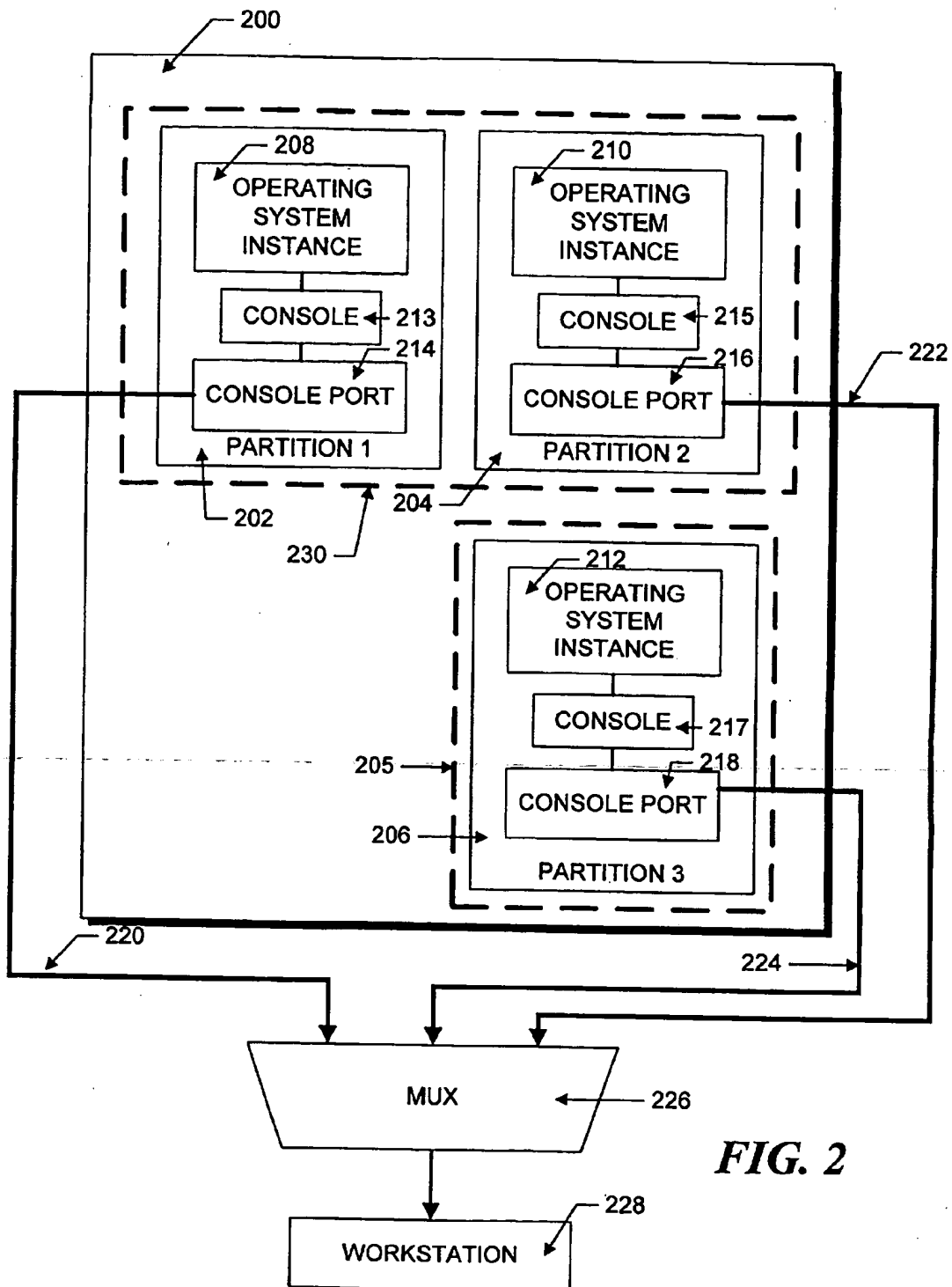
(19) **United States**(12) **Patent Application Publication** (10) Pub. No.: **US 2002/0010844 A1**  
NOEL et al. (43) Pub. Date: **Jan. 24, 2002**(54) **METHOD AND APPARATUS FOR  
DYNAMICALLY SHARING MEMORY IN A  
MULTIPROCESSOR SYSTEM**(52) U.S. Cl. .... 711/153; 711/147; 709/1;  
709/104; 709/106(76) Inventors: **KAREN L. NOEL, PENBROKE, NH  
(US); GREGORY H. JORDAN,  
HOLLIS, NH (US); PAUL K.  
HARTER JR., GROTON, MA (US);  
THOMAS BENSON, HOLLIS, NH  
(US)**Correspondence Address:  
**CONLEY, ROSE & TAYLOR  
P.O. BOX 3267  
HOUSTON, TX 77253-3267 (US)**(\*) Notice: This is a publication of a continued pro-  
secution application (CPA) filed under 37  
CFR 1.53(d).(21) Appl. No.: **09/095,379**(22) Filed: **Jun. 10, 1998****Publication Classification**(51) Int. Cl.<sup>7</sup> ..... **G06F 12/00**(57) **ABSTRACT**

Multiple instances of operating systems execute cooperatively in a single multiprocessor computer wherein all processors and resources are electrically connected together. The single physical machine with multiple physical processors and resources is subdivided by software into multiple partitions, each with the ability to run a distinct copy, or instance, of an operating system. At different times, different operating system instances may be loaded on a given partition. Resources, such as CPUs and memory, can be dynamically assigned to different partitions and used by instances of operating systems running within the machine by modifying the configuration. The partitions themselves can also be changed without rebooting the system by modifying the configuration tree. A grouping of partitions, a community, shares memory. Memory may be private to a particular partition or may be shared by partitions within a community. When a community is formed the creating instance reads a configuration tree and builds management structures for the shared memory owned by the community. A single system may have one or more communities, each with their own representation within the configuration tree.





**FIG. 1**



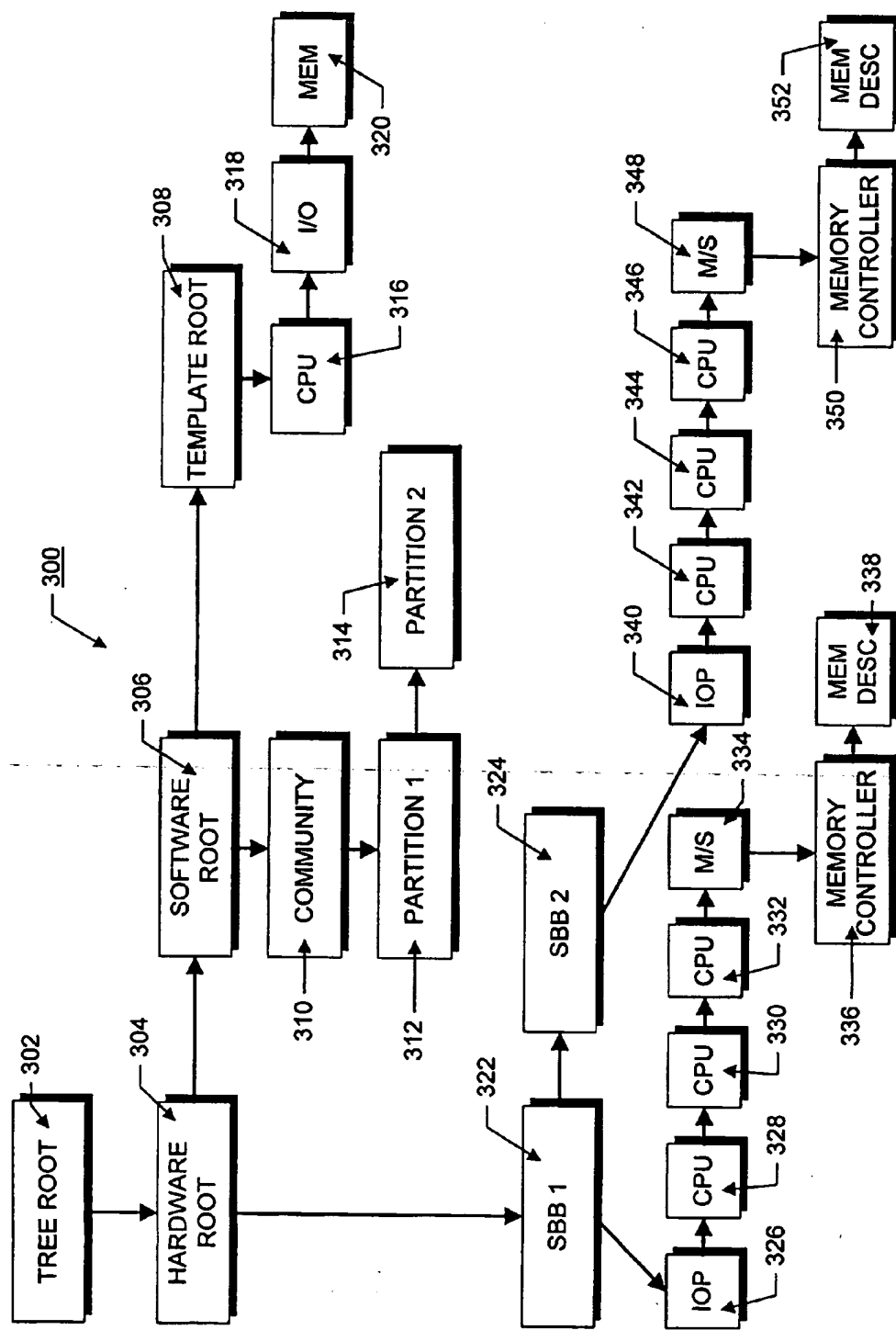


FIG. 3



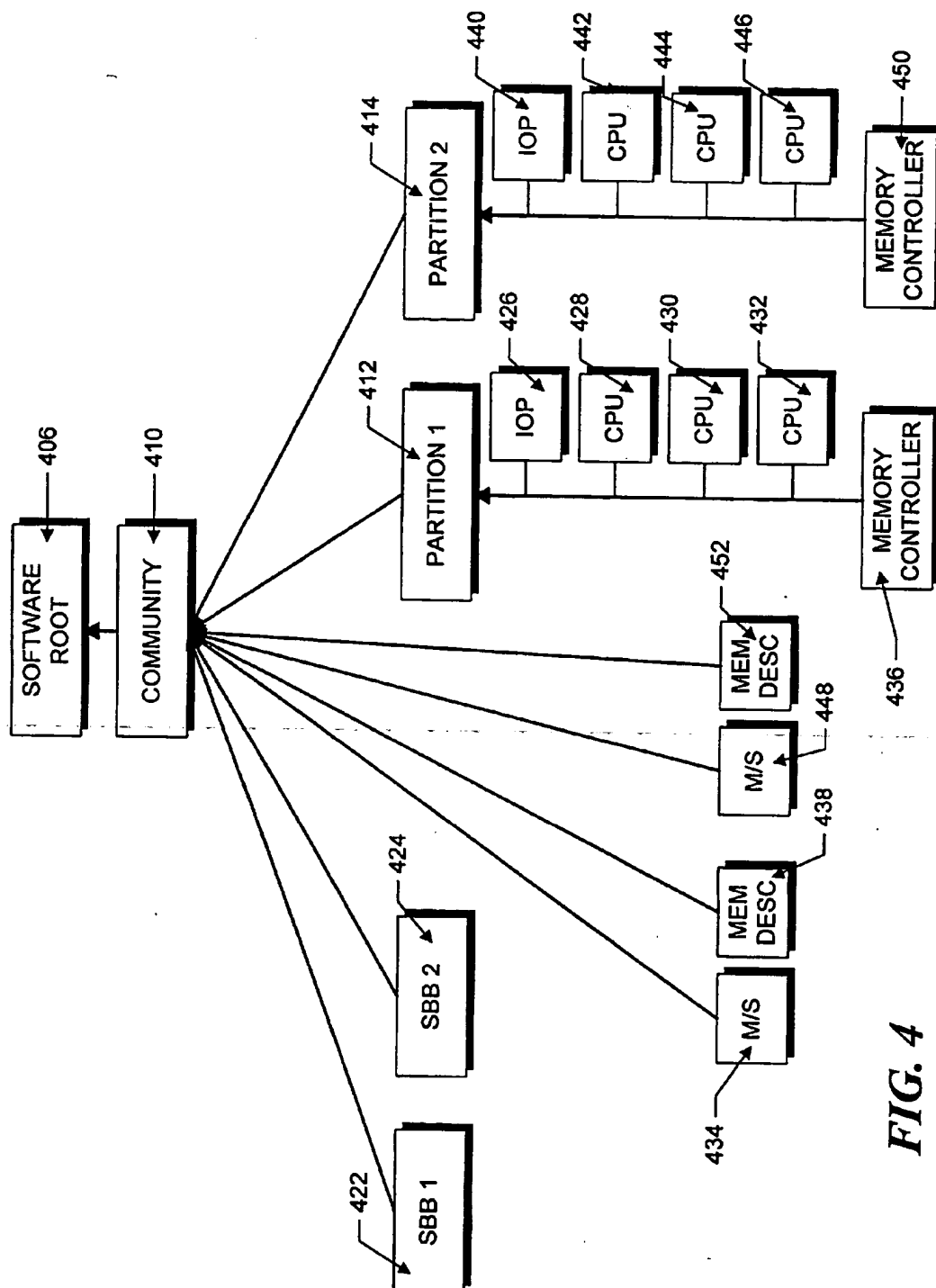
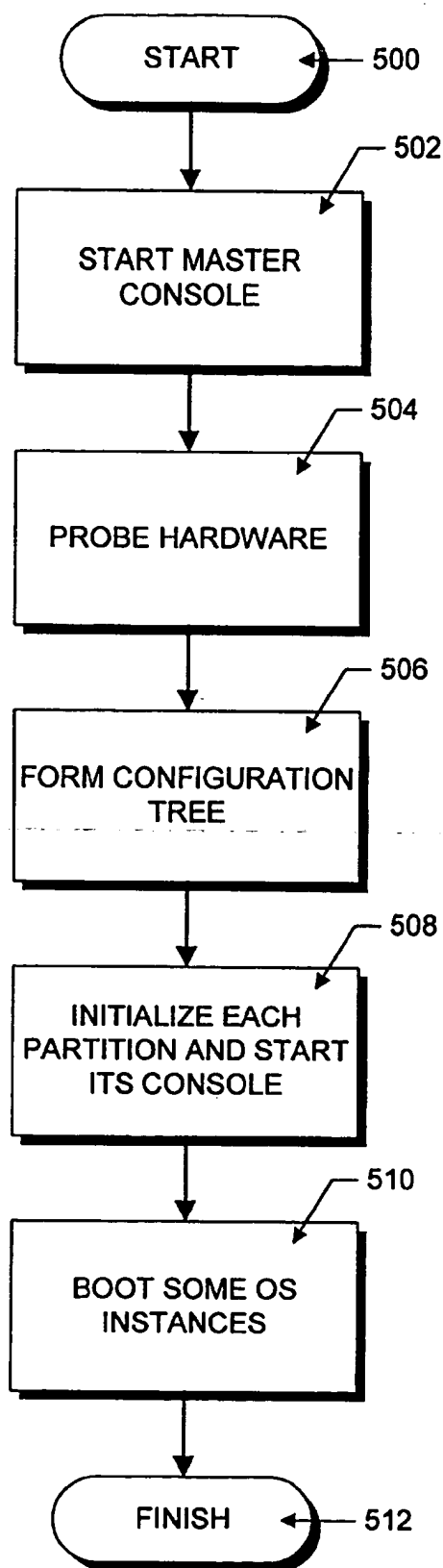
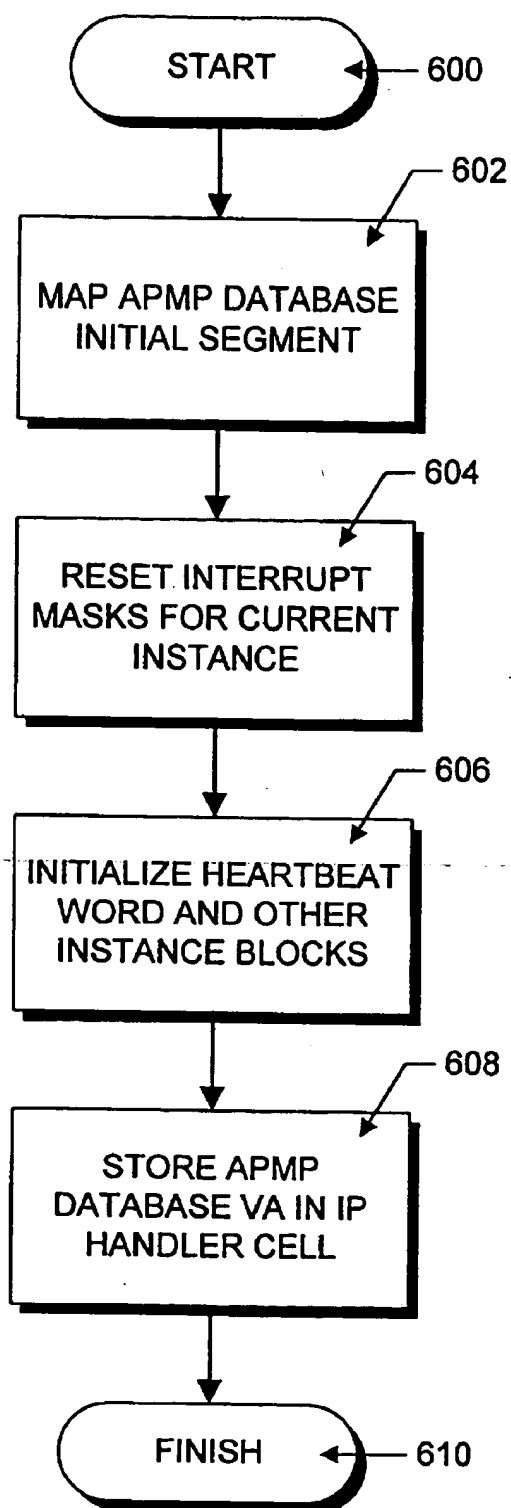


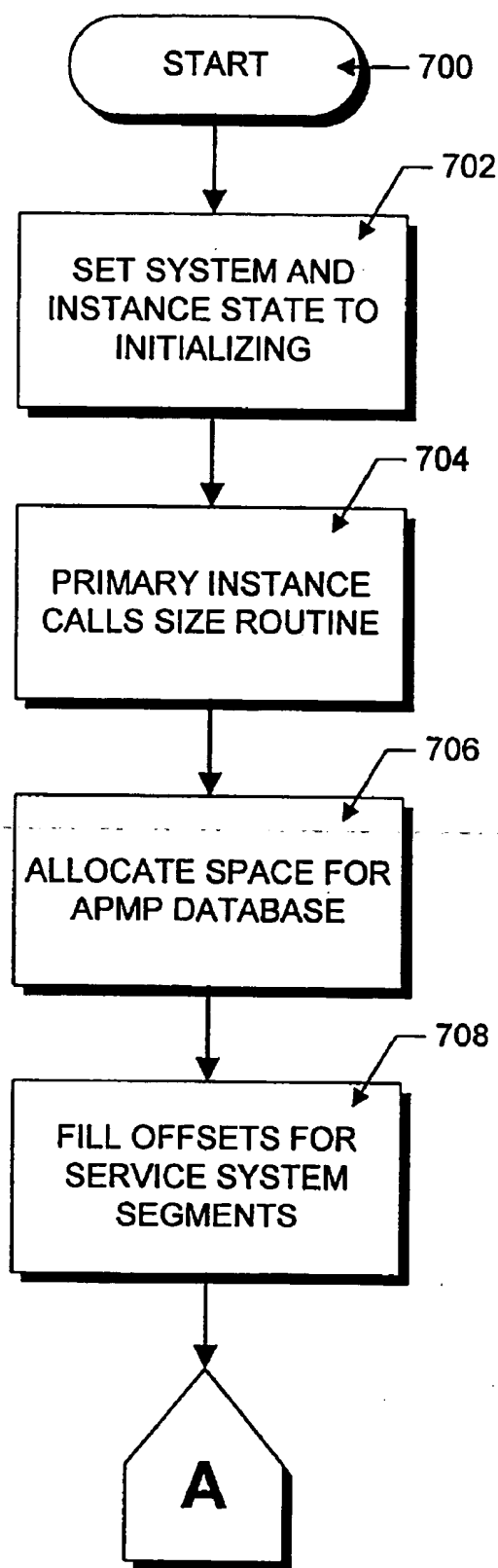
FIG. 4



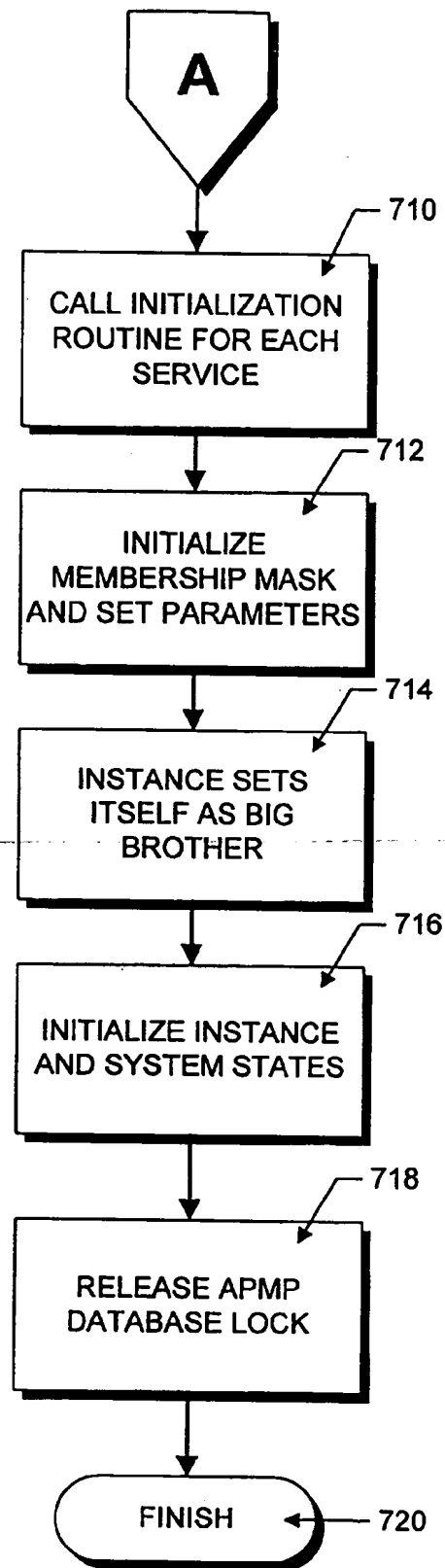
**FIG. 5**



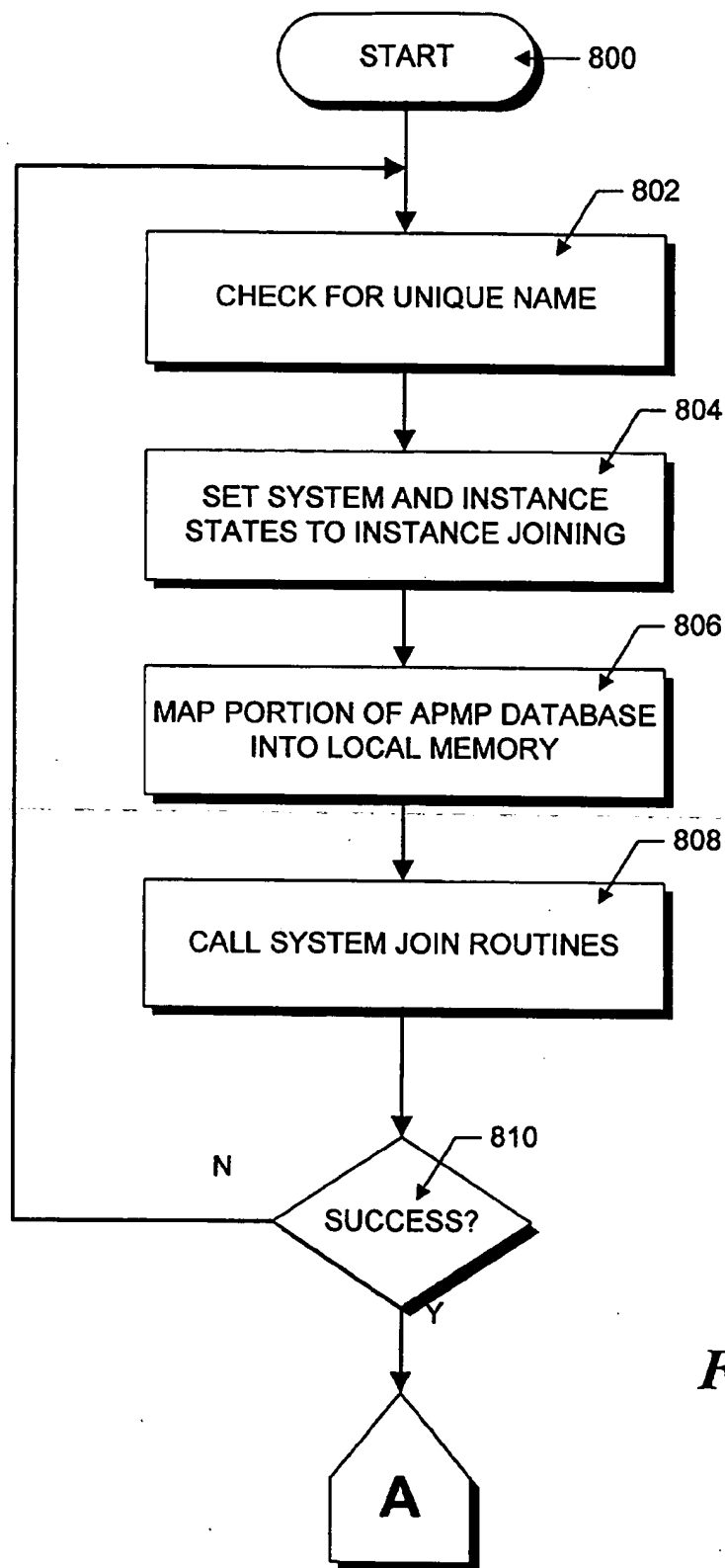
**FIG. 6**



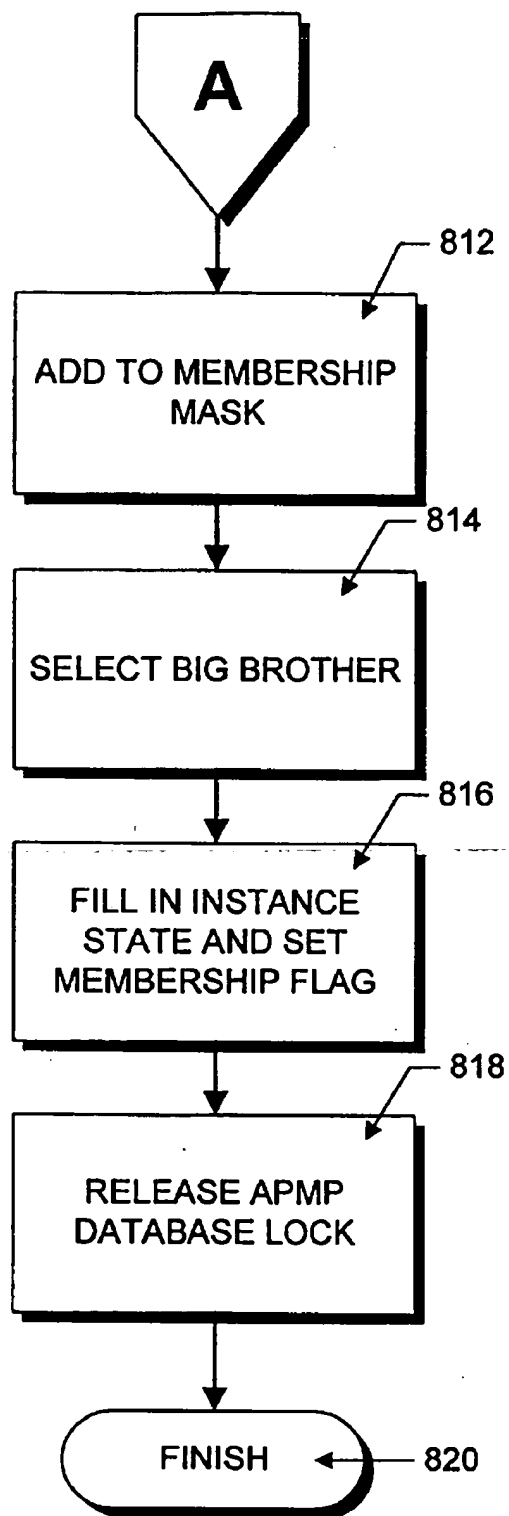
**FIG. 7A**



**FIG. 7B**



**FIG. 8A**



**FIG. 8B**

## METHOD AND APPARATUS FOR DYNAMICALLY SHARING MEMORY IN A MULTIPROCESSOR SYSTEM

### FIELD OF THE INVENTION

[0001] This invention relates to multiprocessor computer architectures in which processors and other computer hardware resources are grouped in partitions, each of which has an operating system instance and, more specifically, to methods and apparatus for allocating computer hardware resources to partitions.

### BACKGROUND OF THE INVENTION

[0002] The efficient operation of many applications in present computing environments depends upon fast, powerful and flexible computing systems. The configuration and design of such systems has become very complicated when such systems are to be used in an "enterprise" commercial environment where there may be many separate departments, many different problem types and continually changing computing needs. Users in such environments generally want to be able to quickly and easily change the capacity of the system, its speed and its configuration. They may also want to expand the system work capacity and change configurations to achieve better utilization of resources without stopping execution of application programs on the system. In addition they may want to be able to configure the system in order to maximize resource availability so that each application will have an optimum computing configuration.

[0003] Traditionally, computing speed has been addressed by using a "shared nothing" computing architecture where data, business logic, and graphic user interfaces are distinct tiers and have specific computing resources dedicated to each tier. Initially, a single central processing unit was used and the power and speed of such a computing system was increased by increasing the clock rate of the single central processing unit. More recently, computing systems have been developed which use several processors working as a team instead one massive processor working alone. In this manner, a complex application can be distributed among many processors instead of waiting to be executed by a single processor. Such systems typically consist of several central processing units (CPUs) which are controlled by a single operating system. In a variant of a multiple processor system called "symmetric multiprocessing" or SMP, the applications are distributed equally across all processors. The processors also share memory. In another variant called "asymmetric multiprocessing" or AMP, one processor acts as a "master" and all of the other processors act as "slaves." Therefore, all operations, including the operating system, must pass through the master before being passed onto the slave processors. These multiprocessing architectures have the advantage that performance can be increased by adding additional processors, but suffer from the disadvantage that the software running on such systems must be carefully written to take advantage of the multiple processors and it is difficult to scale the software as the number of processors increases. Current commercial workloads do not scale well beyond 8-24 CPUs as a single SMP system, the exact number depending upon platform, operating system and application mix.

[0004] For increased performance, another typical answer has been to dedicate computer resources (machines) to an

application in order to optimally tune the machine resources to the application. However, this approach has not been adopted by the majority of users because most sites have many applications and separate databases developed by different vendors. Therefore, it is difficult, and expensive, to dedicate resources among all of the applications especially in environments where the application mix is constantly changing.

[0005] Alternatively, a computing system can be partitioned with hardware to make a subset of the resources on a computer available to a specific application. This approach avoids dedicating the resources permanently since the partitions can be changed, but still leaves issues concerning performance improvements by means of load balancing of resources among partitions and resource availability.

[0006] The availability and maintainability issues were addressed by a "shared everything" model in which a large centralized robust server that contains most of the resources is networked with and services many small, uncomplicated client network computers. Alternatively, "clusters" are used in which each system or "node" has its own memory and is controlled by its own operating system. The systems interact by sharing disks and passing messages among themselves via some type of communications network. A cluster system has the advantage that additional systems can easily be added to a cluster. However, networks and clusters suffer from a lack of shared memory and from limited interconnect bandwidth which places limitations on performance.

[0007] In many enterprise computing environments, it is clear that the two separate computing models must be simultaneously accommodated and each model optimized. Several prior art approaches have been used to attempt this accommodation. For example, a design called a "virtual machine" or VM developed and marketed by International Business Machines Corporation, Armonk, N.Y., uses a single physical machine, with one or more physical processors, in combination with software which simulates multiple virtual machines. Each of those virtual machines has, in principle, access to all the physical resources of the underlying real computer. The assignment of resources to each virtual machine is controlled by a program called a "hypervisor". There is only one hypervisor in the system and it is responsible for all the physical resources. Consequently, the hypervisor, not the other operating systems, deals with the allocation of physical hardware. The hypervisor intercepts requests for resources from the other operating systems and deals with the requests in a globally-correct way.

[0008] The VM architecture supports the concept of a "logical partition" or LPAR. Each LPAR contains some of the available physical CPUs and resources which are logically assigned to the partition. The same resources can be assigned to more than one partition. LPARs are set up by an administrator statically, but can respond to changes in load dynamically, and without rebooting, in several ways. For example, if two logical partitions, each containing ten CPUs, are shared on a physical system containing ten physical CPUs, and, if the logical ten CPU partitions have complementary peak loads, each partition can take over the entire physical ten CPU system as the workload shifts without a re-boot or operator intervention.

[0009] In addition, the CPUs logically assigned to each partition can be turned "on" and "off" dynamically via



normal operating system operator commands without reboot. The only limitation is that the number of CPUs active at system initialization is the maximum number of CPUs that can be turned "on" in any partition.

[0010] Finally, in cases where the aggregate workload demand of all partitions is more than can be delivered by the physical system, LPAR weights can be used to define how much of the total CPU resources is given to each partition. These weights can be changed by operators on-the-fly with no disruption.

[0011] Another prior art system is called a "Parallel Sysplex" and is also marketed and developed by the International Business Machines Corporation. This architecture consists of a set of computers that are clustered via a hardware entity called a "coupling facility" attached to each CPU. The coupling facilities on each node are connected via a fiber-optic link and each node operates as a traditional SMP machine, with a maximum of 10 CPUs. Certain CPU instructions directly invoke the coupling facility. For example, a node registers a data structure with the coupling facility, then the coupling facility takes care of keeping the data structures coherent within the local memory of each node.

[0012] The Enterprise 10000 Unix server developed and marketed by Sun Microsystems, Mountain View, Calif., uses a partitioning arrangement called "Dynamic System Domains" to logically divide the resources of a single physical server into multiple partitions, or domains, each of which operates as a stand-alone server. Each of the partitions has CPUs, memory and I/O hardware. Dynamic reconfiguration allows a system administrator to create, resize, or delete domains on the fly and without rebooting. Every domain remains logically isolated from any other domain in the system, isolating it completely from any software error or CPU, memory, or I/O error generated by any other domain. There is no sharing of resources between any of the domains.

[0013] The Hive Project conducted at Stanford University uses an architecture which is structured as a set of cells. When the system boots, each cell is assigned a range of nodes that it owns throughout execution. Each cell manages the processors, memory and I/O devices on those nodes as if it were an independent operating system. The cells cooperate to present the illusion of a single system to user-level processes.

[0014] Hive cells are not responsible for deciding how to divide their resources between local and remote requests. Each cell is responsible only for maintaining its internal resources and for optimizing performance within the resources it has been allocated. Global resource allocation is carried out by a user-level process called "wax." The Hive system attempts to prevent data corruption by using certain fault containment boundaries between the cells. In order to implement the tight sharing expected from a multiprocessor system despite the fault containment boundaries between cells, resource sharing is implemented through the cooperation of the various cell kernels, but the policy is implemented outside the kernels in the wax process. Both memory and processors can be shared.

[0015] A system called "Cellular IRIX" developed and marketed by Silicon Graphics Inc. Mountain View, Calif.,

supports modular computing by extending traditional symmetric multiprocessing systems. The Cellular IRIX architecture distributes global kernel text and data into optimized SMP-sized chunks or "cells". Cells represent a control domain consisting of one or more machine modules, where each module consists of processors, memory, and I/O. Applications running on these cells rely extensively on a full set of local operating system services, including local copies of operating system text and kernel data structures. Only one instance of the operating system exists on the entire system. Inter-cell coordination allows application images to directly and transparently utilize processing, memory and I/O resources from other cells without incurring the overhead of data copies or extra context switches.

[0016] Another existing architecture called NUMA-Q developed and marketed by Sequent Computer Systems, Inc., Beaverton, Oreg. uses "quads", or a group of four processors per portion of memory, as the basic building block for NUMA-Q SMP nodes. Adding I/O to each quad further improves performance. Therefore, the NUMA-Q architecture not only distributes physical memory but puts a predetermined number of processors and PCI slots next to each part. The memory in each quad is not local memory in the traditional sense. Rather, it is one third of the physical memory address space and has a specific address range. The address map is divided evenly over memory, with each quad containing a contiguous portion of address space. Only one copy of the operating system is running and, as in any SMP system, it resides in memory and runs processes without distinction and simultaneously on one or more processors.

[0017] Accordingly, while many attempts have been made at providing a flexible computer system having maximum resource availability and scalability, existing systems each have significant shortcomings. Therefore, it would be desirable to have a new computer system design which provides improved flexibility, resource availability and scalability.

#### SUMMARY OF THE INVENTION

[0018] In accordance with the principles of the present invention, multiple instances of operating systems execute cooperatively in a single multiprocessor computer wherein all processors and resources are electrically connected together. The single physical machine with multiple physical processors and resources is adaptively subdivided by software into multiple partitions, each with the ability to run a distinct copy, or instance, of an operating system. Each of the partitions has access to its own physical resources plus resources designated as shared. In accordance with one embodiment, the partitioning of resources is performed by assigning resources within a configuration.

[0019] More particularly, software logically, and adaptively, partitions CPUs, memory, and I/O ports by assigning them together. An instance of an operating system may then be loaded on a partition. At different times, different operating system instances may be loaded on a given partition. This partitioning, which a system manager directs, is a software function; no hardware boundaries are required. Each individual instance has the resources it needs to execute independently. Resources, such as CPUs and memory, can be dynamically assigned to different partitions and used by instances of operating systems running within the machine by modifying the configuration. The partitions

themselves can also be changed without rebooting the system by modifying the configuration tree. The resulting adaptively-partitioned, multi-processing (APMP) system exhibits both scalability and high performance.

[0020] The execution environment for a single copy of an operating system, or instance, is referred to as a partition. A community is a grouping of partitions which can share resources. Memory may be private to a particular partition or may be shared by partitions within a community. When an APMP computer system is formed the creating instance reads a configuration tree and builds management structures for the shared resources, including memory, owned by the community. A single system may have one or more communities, each with its own representation within the configuration tree.

[0021] Configuration software selects a group of shared memory pages to be the synchronization point for an APMP computer system. These pages contain information used to determine whether other instances are active members of the APMP computer system. An indication of the location of these synchronization pages is stored within the configuration tree. As an instance joins the APMP computer system, it uses the information within the configuration tree to map to the shared pages. From the contents of the shared pages, the instance can determine whether it is joining an active APMP computer system or it is creating a APMP computer system. If the instance is creating a APMP computer system, it can reconfigure the pages used to synchronize the APMP computer system by modifying the configuration tree. In this way, physical memory that was previously used as a synchronization point may be removed from the system.

[0022] Shared memory may be organized in groups, referred to as common property partitions, according to the hardware characteristics of the memory. Shared memory may be assigned to regions which can be mapped simultaneously by one or more operating system instances. Shared memory may also be mapped by applications running on one or more operating system instances. Shared memory may be "borrowed" by an operating system instance for use as the instance's private memory. Additionally, non uniform memory access is accommodated, in the case of private memory, by allowing the private memory to be owned by the instance whose CPUs have the fastest access to the memory.

#### BRIEF DESCRIPTION OF THE DRAWINGS

[0023] The above and further advantages of the invention may be better understood by referring to the following description in conjunction with the accompanying drawings and which:

[0024] FIG. 1 is a schematic block diagram of a hardware platform illustrating several system building blocks.

[0025] FIG. 2 is a schematic diagram of an APMP computer system constructed in accordance with the principles of the present invention illustrating several partitions.

[0026] FIG. 3 is a schematic diagram of a configuration tree which represents hardware resource configurations and software configurations and their component parts with child and sibling pointers.

[0027] FIG. 4 is a schematic diagram of the configuration tree shown in FIG. 3 and rearranged to illustrate the assignment of hardware to software instances by ownership pointers.

[0028] FIG. 5 is a flowchart outlining steps in an illustrative routine for creating an APMP computer system in accordance with the principles of the present invention.

[0029] FIG. 6 is a flowchart illustrating the steps in an illustrative routine for creating entries in an APMP system management database which maintains information concerning the APMP system and its configuration.

[0030] FIGS. 7A and 7B, when placed together, form a flowchart illustrating in detail the steps in an illustrative routine for creating an APMP computer system in accordance with the principles of the present invention.

[0031] FIGS. 8A and 8B, when placed together, form a flowchart illustrating the steps in an illustrative routine followed by an operating system instance to join an APMP computer system which is already created.

#### DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

[0032] A computer platform constructed in accordance with the principles of the present invention is a multi-processor system capable of being partitioned to allow the concurrent execution of multiple instances of operating system software. The system does not require hardware support for the partitioning of its memory, CPUs and I/O subsystems, but some hardware may be used to provide additional hardware assistance for isolating faults, and minimizing the cost of software engineering. The following specification describes the interfaces and data structures required to support the inventive software architecture. The interfaces and data structures described are not meant to imply a specific operating system must be used, or that only a single type of operating system will execute concurrently. Any operating system which implements the software requirements discussed below can participate in the inventive system operation.

#### [0033] System Building Blocks

[0034] The inventive software architecture operates on a hardware platform which incorporates multiple CPUs, memory and I/O hardware. Preferably, a modular architecture such as that shown in FIG. 1 is used, although those skilled in the art will understand that other architectures can also be used, which architectures need not be modular. FIG. 1 illustrates a computing system constructed of four basic system building blocks (SBBs) 100-106. In the illustrative embodiment, each building block, such as block 100, is identical and comprises several CPUs 108-114, several memory slots (illustrated collectively as memory 120), an I/O processor 118, and a port 116 which contains a switch (not shown) that can connect the system to another such system. However, in other embodiments, the building blocks need not be identical. Large multiprocessor systems can be constructed by connecting the desired number of system building blocks by means of their ports. Switch technology, rather than bus technology, is employed to connect building block components in order to both achieve the improved bandwidth and to allow for non-uniform memory architectures (NUMA).

[0035] In accordance with the principles of the invention, the hardware switches are arranged so that each CPU can address all available memory and I/O ports regardless of the number of building blocks configured as schematically

illustrated by line 122. In addition, all CPUs may communicate to any or all other CPUs in all SBBs with conventional mechanisms, such as inter-processor interrupts. Consequently, the CPUs and other hardware resources can be associated solely with software. Such a platform architecture is inherently scalable so that large amounts of processing power, memory and I/O will be available in a single computer.

[0036] An APMP computer system 200 constructed in accordance with the principles of the present invention from a software view is illustrated in FIG. 2. In this system, the hardware components have been allocated to allow concurrent execution of multiple operating system instances 208, 210, 212.

[0037] In a preferred embodiment, this allocation is performed by a software program called a "console" program, which, as will hereinafter be described in detail, is loaded into memory at power up. Console programs are shown schematically in FIG. 2 as programs 213, 215 and 217. The console program may be a modification of an existing administrative program or a separate program which interacts with an operating system to control the operation of the preferred embodiment. The console program does not virtualize the system resources, that is, it does not create any software layers between the running operating systems 208, 210 and 212 and the physical hardware, such as memory and I/O units (not shown in FIG. 2.) Nor is the state of the running operating systems 208, 210 and 212 swapped to provide access to the same hardware. Instead, the inventive system logically divides the hardware into partitions. It is the responsibility of operating system instance 208, 210, and 212 to use the resources appropriately and provide coordination of resource allocation and sharing. The hardware platform may optionally provide hardware assistance for the division of resources, and may provide fault barriers to minimize the ability of an operating system to corrupt memory, or affect devices controlled by another operating system copy.

[0038] The execution environment for a single copy of an operating system, such as copy 208 is called a "partition" 202, and the executing operating system 208 in partition 202 is called "instance" 208. Each operating system instance is capable of booting and running independently of all other operating system instances in the computer system, and can cooperatively take part in sharing resources between operating system instances as described below.

[0039] In order to run an operating system instance, a partition must include a hardware restart parameter block (HWRPB), a copy of a console program, some amount of memory, one or more CPUs, and at least one I/O bus which must have a dedicated physical port for the console. The HWRPB is a configuration block which is passed between the console program and the operating system.

[0040] Each of console programs 213, 215 and 217, is connected to a console port, shown as ports 214, 216 and 218, respectively. Console ports, such as ports 214, 216 and 218, generally come in the form of a serial line port, or attached graphics, keyboard and mouse options. For the purposes of the inventive computer system, the capability of supporting a dedicated graphics port and associated input devices is not required, although a specific operating system may require it. The base assumption is that a serial port is

sufficient for each partition. While a separate terminal, or independent graphics console, could be used to display information generated by each console, preferably the serial lines 220, 222 and 224, can all be connected to a single multiplexer 226 attached to a workstation, PC, or LAT 228 for display of console information.

[0041] It is important to note that partitions are not synonymous with system building blocks. For example, partition 202 may comprise the hardware in building blocks 100 and 106 in FIG. 1 whereas partitions 204 and 206 might comprise the hardware in building blocks 102 and 104, respectively. Partitions may also include part of the hardware in a building block.

[0042] Partitions can be "initialized" or "uninitialized." An initialized partition has sufficient resources to execute an operating system instance, has a console program image loaded, and a primary CPU available and executing. An initialized partition may be under control of a console program, or may be executing an operating system instance. In an initialized state, a partition has full ownership and control of hardware components assigned to it and only the partition itself may release its components.

[0043] In accordance with the principles of the invention, resources can be reassigned from one initialized partition to another. Reassignment of resources can only be performed by the initialized partition to which the resource is currently assigned. When a partition is in an uninitialized state, other partitions may reassign its hardware components and may delete it.

[0044] An uninitialized partition is a partition which has no primary CPU executing either under control of a console program or an operating system. For example, a partition may be uninitialized due to a lack of sufficient resources at power up to run a primary CPU, or when a system administrator is reconfiguring the computer system. When in an uninitialized state, a partition may reassign its hardware components and may be deleted by another partition. Unassigned resources may be assigned by any partition.

[0045] Partitions may be organized into "communities" which provide the basis for grouping separate execution contexts to allow cooperative resource sharing. Partitions in the same community can share resources. Partitions that are not within the same community cannot share resources. Resources may only be manually moved between partitions that are not in the same community by the system administrator by de-assigning the resource (and stopping usage), and manually reconfiguring the resource. Communities can be used to create independent operating system domains, or to implement user policy for hardware usage. In FIG. 2, partitions 202 and 204 have been organized into community 230. Partition 206 may be in its own community 205. Communities can be constructed using the configuration tree described below and may be enforced by hardware.

[0046] The Console Program

[0047] When a computer system constructed in accordance with the principles of the present invention is enabled on a platform, multiple HWRPB's must be created, multiple console program copies must be loaded, and system resources must be assigned in such a way that each HWRPB is associated with specific components of the system. To do this, the first console program to run will create a configu-

ration tree structure in memory which represents all of the hardware in the system. The tree will also contain the software partitioning information, and the assignments of hardware to partitions and is discussed in detail below.

[0048] More specifically, when the APMP system is powered up, a CPU will be selected as a primary CPU in a conventional manner by hardware which is specific to the platform on which the system is running. The primary CPU then loads a copy of a console program into memory. This console copy is called a "master console" program. The primary CPU initially operates under control of the master console program to perform testing and checking assuming that there is a single system which owns the entire machine. Subsequently, a set of environment variables are loaded which define the system partitions. Finally, the master console creates and initializes the partitions based on the environment variables. In this latter process the master console operates to create the configuration tree, to create additional HWRPB data blocks, to load the additional console program copies, and to start the CPUs on the alternate HWRPBs. Each partition then has an operating system instance running on it, which instance cooperates with a console program copy also running in that partition. In an unconfigured APMP system, the master console program will initially create a single partition containing the primary CPU, a minimum amount of memory, and a physical system administrator's console selected in a platform-specific way. Console program commands will then allow the system administrator to create additional partitions, and configure I/O buses, memory, and CPUs for each partition.

[0049] After associations of resources to partitions have been made by the console program, the associations are stored in non-volatile RAM to allow for an automatic configuration of the system during subsequent boots. During subsequent boots, the master console program must validate the current configuration with the stored configuration to handle the removal and addition of new components. Newly-added components are placed into an unassigned state, until they are assigned by the system administrator. If the removal of a hardware component results in a partition with insufficient resources to run an operating system, resources will continue to be assigned to the partition, but it will be incapable of running an operating system instance until additional new resources are allocated to it.

[0050] As previously mentioned, the console program communicates with an operating system instance by means of an HWRPB which is passed to the operating system during operating system boot up. The fundamental requirements for a console program are that it should be able to create multiple copies of HWRPBs and itself. Each HWRPB copy created by the console program will be capable of booting an independent operating system instance into a private section of memory and each operating system instance booted in this manner can be identified by a unique value placed into the HWRPB. The value indicates the partition, and is also used as the operating system instance ID.

[0051] In addition, the console program is configured to provide a mechanism to remove a CPU from the available CPUs within a partition in response to a request by an operating system running in that partition. Each operating system instance must be able to shutdown, halt, or otherwise

crash in a manner that control is passed to the console program. Conversely, each operating system instance must be able to reboot into an operational mode, independently of any other operating system instance.

[0052] Each HWRPB which is created by a console program will contain a CPU slot-specific database for each CPU that is in the system, or that can be added to the system without powering the entire system down. Each CPU that is physically present will be marked "present", but only CPUs that will initially execute in a specific partition will be marked "available" in the HWRPB for the partition. The operating system instance running on a partition will be capable of recognizing that a CPU may be available at some future time by a present (PP) bit in a per-CPU state flag fields of the HWRPB, and can build data structures to reflect this. When set, the available (PA) bit in the per-CPU state flag fields indicates that the associated CPU is currently associated with the partition, and can be invited to join SMP operation.

#### [0053] The Configuration Tree

[0054] As previously mentioned, the master console program creates a configuration tree which represents the hardware configuration, and the assignment of each component in the system to each partition. Each console program then identifies the configuration tree to its associated operating system instance by placing a pointer to the tree in the HWRPB.

[0055] Referring to FIG. 3, the configuration tree 300 represents the hardware components in the system, the platform constraints and minimums, and the software configuration. The master console program builds the tree using information discovered by probing the hardware, and from information stored in non-volatile RAM which contains configuration information generated during previous initializations.

[0056] The master console may generate a single copy of the tree which copy is shared by all operating system instances, or it may replicate the tree for each instance. A single copy of the tree has the disadvantage that it can create a single point of failure in systems with independent memories. However, platforms that generate multiple tree copies require the console programs to be capable of keeping changes to the tree synchronized.

[0057] The configuration tree comprises multiple nodes including root nodes, child nodes and sibling nodes. Each node is formed of a fixed header and a variable length extension for overlaid data structures. The tree starts with a tree root node 302 representing the entire system box, followed by branches that describe the hardware configuration (hardware root node 304), the software configuration (software root node 306), and the minimum partition requirements (template root node 308.) In FIG. 3, the arrows represent child and sibling relationships. The children of a node represent component parts of the hardware or software configuration. Siblings represent peers of a component that may not be related except by having the same parent. Nodes in the tree 300 contain information on the software communities and operating system instances, hardware configuration, configuration constraints, performance boundaries and hot-swap capabilities. The nodes also provide the relationship of hardware to software ownership, or the sharing of a hardware component.

[0058] The nodes are stored contiguously in memory and the address offset from the tree root node 302 of the tree 300 to a specific node forms a "handle" which may be used from any operating system instance to unambiguously identify the same component on any operating system instance. In addition, each component in the inventive computer system has a separate ID. This may illustratively be a 64-bit unsigned value. The ID must specify a unique component when combined with the type and subtype values of the component. That is, for a given type of component, the ID must identify a specific component. The ID may be a simple number, for example the CPU ID, it may be some other unique encoding, or a physical address. The component ID and handle allow any member of the computer system to identify a specific piece of hardware or software. That is, any partition using either method of specification must be able to use the same specification, and obtain the same result.

[0059] As described above, the inventive computer system is composed of one or more communities which, in turn, are composed of one or more partitions. By dividing the partitions across the independent communities, the inventive computer system can be placed into a configuration in which sharing of devices and memory can be limited. Communities and partitions will have IDs which are densely packed. The hardware platform will determine the maximum number of partitions based on the hardware that is present in the system, as well as having a platform maximum limit. Partition and community IDs will never exceed this value during runtime. IDs will be reused for deleted partitions and communities. The maximum number of communities is the same as the maximum number of partitions. In addition, each operating system instance is identified by a unique instance identifier, for example a combination of the partition ID plus an incarnation number.

[0060] The communities and partitions are represented by a software root node 306, which has community node children (of which community node 310 is shown), and partition node grandchildren (of which two nodes, 312 and 314, are shown.) The hardware components are represented by a hardware root node 304 which contains children that represent a hierarchical representation of all of the hardware currently present in the computer system. "Ownership" of a hardware component is represented by a handle in the associated hardware node which points to the appropriate software node (310, 312 or 314.) These handles are illustrated in FIG. 4 which will be discussed in more detail below. Components that are owned by a specific partition will have handles that point to the node representing the partition. Hardware which is shared by multiple partitions (for example, memory) will have handles that point to the community to which sharing is confined. Un-owned hardware will have a handle of zero (representing the tree root node 302).

[0061] Hardware components place configuration constraints on how ownership may be divided. A "config" handle in the configuration tree node associated with each component determines if the component is free to be associated anywhere in the computer system by pointing to the hardware root node 304. However, some hardware components may be bound to an ancestor node and must be configured as part of this node. Examples of this are CPUs, which may have no constraints on where they execute, but which are a component part of a system building block

(SBB), such as SBBs 322 or 324. In this case, even though the CPU is a child of the SBB, its config handle will point to the hardware root node 304. An I/O bus, however, may not be able to be owned by a partition other than the partition that owns its I/O processor. In this case, the configuration tree node representing the I/O bus would have a config handle pointing to the I/O processor. Because the rules governing hardware configuration are platform specific, this information is provided to the operating system instances by the config handle.

[0062] Each hardware component also has an "affinity" handle. The affinity handle is identical to the config handle, except that it represents a configuration which will obtain the best performance of the component. For example, a CPU or memory may have a config handle which allows it to be configured anywhere in the computer system (it points to the hardware root node 304), however, for optimal performance, the CPU or memory should be configured to use the System Building Block of which they are a part. The result is that the config pointer points to the hardware root node 304, but the affinity pointer points to an SBB node such as node 322 or node 324. The affinity of any component is platform specific, and determined by the firmware. Firmware may use this information when asked to form "optimal" automatic configurations.

[0063] Each node also contains several flags which indicate the type and state of the node. These flags include a node\_hotswap flag which indicates that the component represented is a "hot swappable" component and can be powered down independently of its parent and siblings. However, all children of this node must power down if this component powers down. If the children can power down independently of this component, they must also have this bit set in their corresponding nodes. Another flag is a node\_unavailable flag which, when set, indicates that the component represented by the node is not currently available for use. When a component is powered down (or is never powered up) it is flagged as unavailable.

[0064] Two flags, node hardware and node\_template, indicate the type of node. Further flags, such as node\_initialized and node\_cpu\_primary may also be provided to indicate whether the node represents a partition which has been initialized or a CPU that is currently a primary CPU.

[0065] The configuration tree 300 may extend to the level of device controllers, which will allow the operating system to build bus and device configuration tables without probing the buses. However, the tree may also end at any level, if all components below it cannot be configured independently. System software will still be required to probe for bus and device information not provided by the tree.

[0066] The console program implements and enforces configuration constraints, if any, on each component of the system. In general, components are either assignable without constraints (for example, CPUs may have no constraints), or are configurable only as a part of another component (a device adapter, for example, may be configurable only as a part of its bus). A partition which is, as explained above, a grouping of CPUs, memory, and I/O devices into a unique software entity also has minimum requirements. For example, the minimum hardware requirements for a partition are at least one CPU, some private memory (platform dependent minimum, including console memory) and an I/O bus, including a physical, non-shared, console port.

[0067] The minimal component requirements for a partition are provided by the information contained in the template root node 308. The template root node 308 contains nodes, 316, 318 and 320, representing the hardware components that must be provided to create a partition capable of execution of a console program and an operating system instance. Configuration editors can use this information as the basis to determine what types, and how many resources must be available to form a new partition.

[0068] During the construction of a new partition, the template subtree will be "walked", and, for each node in the template subtree, there must be a node with the same type and subtype owned by the new partition so that it will be capable of loading a console program and booting an operating system instance. If there are more than one node of the same type and subtype in the template tree, there must also be multiple nodes in the new partition. The console program will use the template to validate that a new partition has the minimum requirements prior to attempting to load a console program and initialize operation.

[0069] The following is a detailed example of a particular implementation of configuration tree nodes. It is intended for descriptive purposes only and is not intended to be limiting. Each HWRPB must point to a configuration tree which provides the current configuration, and the assignments of components to partitions. A configuration pointer (in the CONFIG field) in the HWRPB is used to point to the configuration tree. The CONFIG field points to a 64-byte header containing the size of the memory pool for the tree, and the initial checksum of the memory. Immediately following the header is the root node of the tree. The header and root node of the tree will be page aligned.

[0070] The total size in bytes of the memory allocated for the configuration tree is located in the first quadword of the header. The size is guaranteed to be in multiples of the hardware page size. The second quadword of the header is reserved for a checksum. In order to examine the configuration tree, an operating system instance maps the tree into its local address space. Because an operating system instance may map this memory with read access allowed for all applications, some provision must be made to prevent a non-privileged application from gaining access to console data to which it should not have access. Access may be restricted by appropriately allocating memory. For example, the memory may be page aligned and allocated in whole pages. Normally, an operating system instance will map the first page of the configuration tree, obtain the tree size, and then remap the memory allocated for configuration tree usage. The total size may include additional memory used by the console for dynamic changes to the tree.

[0071] Preferably, configuration tree nodes are formed with fixed headers, and may optionally contain type-specific information following the fixed portion. The size field contains the full length of the node, nodes are illustratively allocated in multiples of 64-bytes and padded as needed. The following description defines illustrative fields in the fixed header for a node:

```
[0072] typedef struct_gct_node {
    [0073] unsigned char type;
    [0074] unsigned char subtype;
    [0075] uint16 size;
    [0076] GCT_HANDLE owner;
```

```
    [0077] GCT_HANDLE current_owner;
    [0078] GCT_ID id;
    [0079] union {
    [0080] uint64 node_flags;
    [0081] struct {
    [0082] unsigned node_hardware:1;
    [0083] unsigned node_hotswap:1;
    [0084] unsigned node_unavailable:1;
    [0085] unsigned node_hw_template:1;
    [0086] unsigned node_initialized:1;
    [0087] unsigned node_cpu_primary:1;
    [0088] #define NODE_HARDWARE 0x001
    [0089] #define NODE_HOTSWAP 0x002
    [0090] #define NODE_UNAVAILABLE 0x004
    [0091] #define NODE_HW_TEMPLATE 0x008
    [0092] #define NODE_INITIALIZED 0x010
    [0093] #define NODE_PRIMARY 0x020
    [0094] } flag_bits;
    [0095] } flag_union;
    [0096] GCT_HANDLE config;
    [0097] GCT_HANDLE affinity;
    [0098] GCT_HANDLE parent;
    [0099] GCT_HANDLE next_sib;
    [0100] GCT_HANDLE prev_sib;
    [0101] GCT_HANDLE child;
    [0102] GCT_HANDLE reserved;
    [0103] uint32 magic
    [0104] } GCT_NODE;
```

[0105] In the above definition the type definitions "uint" are unsigned integers with the appropriate bit lengths. As previously mentioned, nodes are located and identified by a handle (identified by the typedef GCT\_HANDLE in the definition above). An illustrative handle is a signed 32-bit offset from the base of the configuration tree to the node. The value is unique across all partitions in the computer system. That is, a handle obtained on one partition must be valid to lookup a node, or as an input to a console callback, on all partitions. The magic field contains a predetermined bit pattern which indicates that the node is actually a valid node.

[0106] The tree root node represents the entire system. Its handle is always zero. That is, it is always located at the first physical location in the memory allocated for the configuration tree following the config header. It has the following definition:

```
[0107] typedef struct_gct_root_node {
    [0108] GCT_NODE hd;
    [0109] uint64 lock;
    [0110] uint64 transient_level;
    [0111] uint64 current_level;
```

- [0112] uint64 console\_req;
- [0113] uint64 min\_alloc;
- [0114] uint64 min\_align;
- [0115] uint64 base\_alloc;
- [0116] uint64 base\_align;
- [0117] uint64 max\_phys\_address;
- [0118] uint64 mem\_size;
- [0119] uint64 platform\_type;
- [0120] int32 platform\_name;
- [0121] GCT\_HANDLE primary\_instance;
- [0122] GCT\_HANDLE first\_free;
- [0123] GCT\_HANDLE high\_limit;
- [0124] GCT\_HANDLE lookaside;
- [0125] GCT\_HANDLE available;
- [0126] uint32 max\_partition;
- [0127] int32 partitions;
- [0128] int32 communities;
- [0129] uint32 max\_platform\_partition;
- [0130] uint32 max\_fragments;
- [0131] uint32 max\_desc;
- [0132] char APMP\_id[16];
- [0133] char APMP\_id\_pad[4];
- [0134] int32 bindings;
- [0135] { GCT\_ROOT\_NODE;
- [0136] The fields in the root node are defined as follows:
- [0137] lock
 

[0138] This field is used as a simple lock by software wishing to inhibit changes to the structure of the tree, and the software configuration. When this value is -1 (all bits on) the tree is unlocked; when the value is  $\geq 0$  the tree is locked. This field is modified using atomic operations. The caller of the lock routine passes a partition ID which is written to the lock field. This can be used to assist in fault tracing, and recovery during crashes.
- [0139] transient\_level
- [0140] This field is incremented at the start of a tree update.
- [0141] current\_level
- [0142] This field is updated at the completion of a tree update.
- [0143] console\_req
- [0144] This field specifies the memory required in bytes for the console in the base memory segment of a partition.
- [0145] min\_alloc
- [0146] This field holds the minimum size of a memory fragment, and the allocation unit (fragments size must be a multiple of the allocation). It must be a power of 2.
- [0147] min\_align
- [0148] This field holds the alignment requirements for a memory fragment. It must be a power of 2.
- [0149] base\_alloc
- [0150] This field specifies the minimum memory in bytes (including console\_req) needed for the base memory segment for a partition. This is where the console, console structures, and operating system will be loaded for a partition. It must be greater or equal to minAlloc and a multiple of minAlloc.
- [0151] base\_align
- [0152] This field holds the alignment requirement for the base memory segment of a partition. It must be a power of 2, and have an alignment of at least min\_align.
- [0153] max\_phys\_address
- [0154] The field holds the calculated largest physical address that could exist on the system, including memory subsystems that are not currently powered on and available.
- [0155] mem\_size
- [0156] This field holds the total memory currently in system.
- [0157] platform\_type
- [0158] This field stores the type of platform taken from a field in the HWRPB.
- [0159] platform\_name
- [0160] This field holds an integer offset from the base of the tree root node to a string representing the name of the platform.
- [0161] primary\_instance
- [0162] This field stores the partition ID of the first operating system instance.
- [0163] first\_free
- [0164] This field holds the offset from the tree root node to the first free byte of memory pool used for new nodes.
- [0165] high\_limit
- [0166] This field holds the highest address at which a valid node can be located within the configuration tree. It is used by callbacks to validate that a handle is legal.
- [0167] lookaside
- [0168] This field is the handle of a linked list of nodes that have been deleted, and that may be reclaimed. When a community or partition are deleted, the node is linked into this list, and creation of a new partition or community will look at this list before allocating from free pool.
- [0169] available
- [0170] This field holds the number of bytes remaining in the free pool pointed to by the first free field.
- [0171] max\_partitions
- [0172] This field holds the maximum number of partitions computed by the platform based on the amount of hardware resources currently available.

**[0173]** partitions

**[0174]** This field holds an offset from the base of the root node to an array of handles. Each partition ID is used as an index into this array, and the partition node handle is stored at the indexed location. When a new partition is created, this array is examined to find the first partition ID which does not have a corresponding partition node handle and this partition ID is used as the ID for the new partition.

**[0175]** communities

**[0176]** This field also holds an offset from the base of the root node to an array of handles. Each community ID is used as an index into this array, and a community node handle is stored in the array. When a new community is created, this array is examined to find the first community ID which does not have a corresponding community node handle and this community ID is used as the ID for the new community. There cannot be more communities than partitions, so the array is sized based on the maximum number of partitions.

**[0177]** max\_platform\_partition

**[0178]** This field holds the maximum number of partitions that can simultaneously exist on the platform, even if additional hardware is added (potentially in-swapped).

**[0179]** max\_fragments

**[0180]** This field holds a platform defined maximum number of fragments into which a memory descriptor can be divided. It is used to size the array of fragments in the memory descriptor node.

**[0181]** max\_desc

**[0182]** This field holds the maximum number of memory descriptors for the platform.

**[0183]** APMP\_id

**[0184]** This field holds a system ID set by system software and saved in non-volatile RAM.

**[0185]** APMP\_id\_pad

**[0186]** This field holds padding bytes for the APMP ID.

**[0187]** bindings

**[0188]** This field holds an offset to an array of "bindings". Each binding entry describes a type of hardware node, the type of node the parent must be, the configuration binding, and the affinity binding for a node type. Bindings are used by software to determine how node types are related and configuration and affinity rules.

**[0189]** A community provides the basis for the sharing of resources between partitions. While a hardware component may be assigned to any partition in a community, the actual sharing of a device, such as memory, occurs only within a community. The community node 310 contains a pointer to a control section, called an APMP database, which allows the operating system instances to control access and membership in the community for the purpose of sharing memory and communications between instances. The APMP database and the creation of communities are discussed in detail below. The configuration ID for the community is a signed 16-bit integer value assigned by the console program. The ID value will never be greater than the maximum number of partitions that can be created on the platform.

**[0190]** A partition node, such as node 312 or 314, represents a collection of hardware that is capable of running an independent copy of the console program, and an independent copy of an operating system. The configuration ID for this node is a signed 16-bit integer value assigned by the console. The ID will never be greater than the maximum number of partitions that can be created on the platform. The node has the definition:

**[0191]** typedef struct \_gct\_partition\_node

**[0192]** GCT\_NODE hd;

**[0193]** uint64 hwrpb;

**[0194]** uint64 incarnation;

**[0195]** uint64 priority;

**[0196]** int32 os\_type;

**[0197]** uint32 partition\_reserved\_1;

**[0198]** uint64 instance\_name\_format;

**[0199]** char instance\_name[128];

**[0200]** } GCT\_PARTITION\_NODE;

**[0201]** The defined fields have the definitions:

**[0202]** hwrpb

**[0203]** This field holds the physical address of the hardware restart parameter block for this partition. To minimize changes to the HWRPB, the HWRPB does not contain a pointer to the partition, or the partition ID. Instead, the partition nodes contain a pointer to the HWRPB. System software can then determine the partition ID of the partition in which it is running by searching the partition nodes for the partition which contains the physical address of its HWRPB.

**[0204]** incarnation

**[0205]** This field holds a value which is incremented each time the primary CPU of the partition executes a boot or restart operation on the partition.

**[0206]** priority

**[0207]** This field holds a partition priority.

**[0208]** os\_type

**[0209]** This field holds a value which indicates the type of operating system that will be loaded in the partition.

**[0210]** partition\_reserved\_1

**[0211]** This field is reserved for future use.

**[0212]** instance\_name\_format

**[0213]** This field holds a value that describes the format of the instance name string.

**[0214]** instance\_name

**[0215]** This field holds a formatted string which is interpreted using the instance\_name\_format field. The value in this field provides a high-level path name to the operating system instance executing in the partition. This field is loaded by system software and is not saved across power cycles. The field is cleared at power up and at partition creation and deletion.



[0216] A System Building Block node, such as node 322 or 324, represents an arbitrary piece of hardware, or conceptual grouping used by system platforms with modular designs such as that illustrated in FIG. 2. A QBB (Quad Building Block) is a specific example of an SBB and corresponds to units such as units 100, 102, 104 and 106 in FIG. 1. Children of the SBB nodes 322 and 324 include input/output processor nodes 326 and 340.

[0217] CPU nodes, such as nodes 328-332 and 342-346, are assumed to be capable of operation as a primary CPU for SMP operation. In the rare case where a CPU is not primary capable, it will have a SUBTYPE code indicating that it cannot be used as a primary CPU in SMP operation. This information is critical when configuring resources to create a new partition. The CPU node will also carry information on where the CPU is currently executing. The primary for a partition will have the NODE\_CPU\_PRIMARY flag set in the NODE\_FLAGS field. The CPU node has the following definition:

[0218] typedef struct gct\_cpu\_node

[0219] GCT\_NODE hd;

[0220] } GCT\_CPU\_NODE;

[0221] A memory subsystem node, such as node 334 or 348, is a "pseudo" node that groups together nodes representing the physical memory controllers and the assignments of the memory that the controllers provide. The children of this node consist of one or more memory controller nodes (such as nodes 336 and 350) which the console has configured to operate together (interleaved), and one or more memory descriptor nodes (such as nodes 338 and 352) which describe physically contiguous ranges of memory.

[0222] A memory controller node (such as nodes 336 or 350) is used to express a physical hardware component, and its owner is typically the partition which will handle errors, and initialization. Memory controllers cannot be assigned to communities, as they require a specific operating system instance for initialization, testing and errors. However, a memory description, defined by a memory descriptor node, may be split into "fragments" to allow different partitions or communities to own specific memory ranges within the memory descriptor. Memory is unlike other hardware resources in that it may be shared concurrently, or broken into "private" areas. Each memory descriptor node contains a list of subset ranges that allow the memory to be divided among partitions, as well as shared between partitions (owned by a community). A memory descriptor node (such as nodes 338 or 352) is defined as:

[0223] typedef struct gct\_mem\_desc\_node {

[0224] GCT\_NODE hd;

[0225] GCT\_MEM\_INFO mem\_info;

[0226] int32 mem\_frag;

[0227] } GCT\_MEM\_DESC\_NODE;

[0228] The mem\_info structure has the following definition:

[0229] typedef struct gctmem\_info {

[0230] uint64 base\_pa;

[0231] uint64 base\_size;

[0232] uint32 desc\_count;

[0233] uint32 info\_fill;

[0234] } GCT\_MEM\_INFO;

[0235] The mem\_frag field holds an offset from the base of the memory descriptor node to an array of GCT\_MEM\_DESC structures which have the definition:

[0236] typedef struct gct\_mem\_desc {

[0237] uint64 pa;

[0238] uint64 size;

[0239] GCT\_HANDLE mem\_owner;

[0240] GCT\_HANDLE mem\_current\_owner;

[0241] union{

[0242] uint32 mem\_flags;

[0243] struct{

[0244] unsigned mem\_console:1;

[0245] unsigned mem\_private:1;

[0246] unsigned mem\_shared:1;

[0247] unsigned base:1;

[0248] #define CGT\_MEM\_CONSOLE 0x1

[0249] #define CGT\_MEM\_PRIVATE 0x2

[0250] #define CGT\_MEM\_SHARED 0x4

[0251] #define CGT\_MEM\_CONSOLE 0x8

[0252] } flag\_bits;

[0253] } flag\_union;

[0254] uint32 mem\_fill;

[0255] } GCT\_MEM\_DESC;

[0256] The number of fragments in a memory description node (nodes 338 or 352) is limited by platform firmware. This creates an upper bound on memory division, and limits unbounded growth of the configuration tree. Software can determine the maximum number of fragments from the max\_fragments field in the tree root node 302 (discussed above), or by calling an appropriate console callback function to return the value. Each fragment can be assigned to any partition, provided that the config binding, and the ownership of the memory descriptor and memory subsystem nodes allow it. Each fragment contains a base physical address, size, and owner field, as well as flags indicating the type of usage.

[0257] To allow shared memory access, the memory subsystem parent node, and the memory descriptor node must be owned by a community. The fragments within the memory descriptor may then be owned by the community (shared) or by any partition within the community.

[0258] Fragments can have minimum allocation sizes and alignments provided in the tree root node 302. The base memory for a partition (the fragments where the console and operating system will be loaded) may have a greater allocation and alignment than other fragments (see the tree root

node definition above). If the owner field of the memory descriptor node is a partition, then the fragments can only be owned by that partition.

[0259] FIG. 4 illustrates the configuration tree shown in FIG. 3 when it is viewed from a perspective of ownership. The console program for a partition relinquishes ownership and control of the partition resources to the operating system instance running in that partition when the primary CPU for that partition starts execution. The concept of "ownership" determines how the hardware resources and CPUs are assigned to software partitions and communities. The configuration tree has ownership pointers illustrated in FIG. 4 which determine the mapping of hardware devices to software such as partitions (exclusive access) and communities (shared access). An operating system instance uses the information in the configuration tree to determine to which hardware resources it has access and reconfiguration control.

[0260] Passive hardware resources which have no owner are unavailable for use until ownership is established. Once ownership is established by altering the configuration tree, the operating system instances may begin using the resources. When an instance makes an initial request, ownership can be changed by causing the owning operating system to stop using a resource or by a console program taking action to stop using a resource in a partition where no operating system instance is executing. The configuration tree is then altered to transfer ownership of the resource to another operating system instance. The action required to cause an operating system to stop using a hardware resource is operating system specific, and may require a reboot of the operating system instances affected by the change.

[0261] To manage the transition of a resource from an owned and active state, to a unowned and inactive state, two fields are provided in each node of the tree. The owner field represents the owner of a resource and is loaded with the handle of the owning software partition or community. At power up of an APMP system, the owner fields of the hardware nodes are loaded from the contents of non-volatile RAM to establish an initial configuration.

[0262] To change the owner of a resource, the handle value is modified in the owner field of the hardware component, and in the owner fields of any descendants of the hardware component which are bound to the component by their config handles. The current\_owner field represents the current user of the resource. When the owner and current\_owner fields hold the same non-zero value, the resource is owned and active. Only the owner of a resource can deassign the resource (set the owner field to zero). A resource that has null owner and current\_owner fields is unowned, and inactive. Only resources which have null owner and current\_owner fields may be assigned to a new partition or community.

[0263] When a resource is de-assigned, the owner may decide to deassign the owner field, or both the owner and current\_owner fields. The decision is based on the ability of the owning operating system instance running in the partition to discontinue the use of the resource prior to de-assigning ownership. In the case where a reboot is required to relinquish ownership, the owner field is cleared, but the current\_owner field is not changed. When the owning operating system instance reboots, the console program can clear any current\_owner fields for resources that have no owner during initialization.

[0264] During initialization, the console program will modify the current\_owner field to match the owner field for any node of which it is the owner, and for which the current\_owner field is null. System software should only use hardware of which it is the current owner. In the case of a de-assignment of a resource which is owned by a community, it is the responsibility of system software to manage the transition between states. In some embodiments, a resource may be loaned to another partition. In this condition, the owner and current\_owner fields are both valid, but not equal. The following table summarizes the possible resource states and the values of the owner and current\_owner fields:

TABLE 1

owner field value	current owner field value	Resource State
none	none	unowned, and inactive
none	valid	unowned, but still active
valid	none	owned, not yet active
valid	equal to owner	owned and active
valid	is not equal to owner	loaned

[0265] Because CPUs are active devices, and sharing of CPUs means that a CPU could be executing in the context of a partition which may not be its "owner", ownership of a CPU is different from ownership of a passive resource. The CPU node in the configuration tree provides two fields that indicate which partition a CPU is nominally "owned" by, and in which partition the CPU is currently executing. The owner field contains a value which indicates the nominal ownership of the CPU, or more specifically, the partition in which the CPU will initially execute at system power up.

[0266] Until an initial ownership is established (that is, if the owner field is unassigned), CPUs are placed into a HWRPB context decided by the master console, but the HWRPB available bit for the CPU will not be set in any HWRPB. This combination prevents the CPU from joining any operating system instance in SMP operation. When ownership of a CPU is established (the owner field is filled in with a valid partition handle), the CPU will migrate, if necessary, to the owning partition, set the available bit in the HWRPB associated with that partition, and request to join SMP operation of the instance running in that partition, or join the console program in SMP mode. The combination of the present and available bits in the HWRPB tell the operating system instance that the CPU is available for use in SMP operation, and the operating system instance may use these bits to build appropriate per-CPU data structures, and to send a message to the CPU to request it to join SMP operation.

[0267] When a CPU sets the available bit in an HWRPB, it also enters a value into the current\_owner field in its corresponding CPU node in the configuration tree. The current\_owner field value is the handle of the partition in which the CPU has set the active HWRPB bit and is capable of joining SMP operation. The current\_owner field for a CPU is only set by the console program. When a CPU migrates from one partition to another partition, or is halted into an unassigned state, the current\_owner field is cleared (or changed to the new partition handle value) at the same time that the available bit is cleared in the HWRPB. The current owner field should not be written to directly by system software, and only reflects which HWRPB has the available bit set for the CPU.

[0268] During runtime, an operating system instance can temporarily "loan" a CPU to another partition without changing the nominal ownership of the CPU. The traditional SMP concept of ownership using the HWRPB present and available bits is used to reflect the current execution context of the CPU by modifying the HWRPB and the configuration tree in atomic operations. The current\_owner field can further be used by system software in one of the partitions to determine in which partition the CPU is currently executing (other instances can determine the location of a particular CPU by examining the configuration tree.)

[0269] It is also possible to de-assign a CPU and return it into a state in which the available bit is not set in any HWRPB, and the current\_owner field in the configuration tree node for the CPU is cleared. This is accomplished by halting the execution of the CPU and causing the console program to clear the owner field in the configuration tree node, as well as the current\_owner field and the available HWRPB bit. The CPU will then execute in console mode and poll the owner field waiting for a valid partition handle to be written to it. System software can then establish a new owner, and the CPU begin execution in the new partition.

[0270] Illustrative ownership pointers are illustrated in FIG. 4 by arrows. Each of the nodes in FIG. 4 that corresponds to a similar node in FIG. 3 is given a corresponding number. For example, the software root node denoted in FIG. 3 as node 306 is denoted as node 406 in FIG. 4. As shown in FIG. 4, the community 410 is "owned" by the software root 406. Likewise, the system building blocks 1 and 2 (422 and 425) are owned by the community 410. Similarly, partitions 412 and 414 are also owned by the community 410.

[0271] Partition 412 owns CPU 428-432 and the I/O processor 426. The memory controller 436 is also a part of partition 1 (412). In a like manner, partition 2 (414) owns CPUs 442-446, I/O processor 440 and memory controller 450.

[0272] The common or shared memory in the system is comprised of memory subsystems 434 and 448 and memory descriptors 438 and 452. These are owned by the community 410. Thus, FIG. 4 describes the layout of the system as it would appear to the operating system instances.

#### [0273] Operating System Characteristics

[0274] As previously mentioned, the illustrative computer system can operate with several different operating systems in different partitions. However, conventional operating systems may need to be modified in some aspects in order to make them compatible with the inventive system, depending on how the system is configured. Some sample modifications for the illustrative embodiment are listed below:

[0275] 1. Instances may need to be modified to include a mechanism for choosing a "primary" CPU in the partition to run the console and be a target for communication from other instances. The selection of a primary CPU can be done in a conventional manner using arbitration mechanisms or other conventional devices.

[0276] 2. Each instance may need modifications that allow it to communicate and cooperate with the console program which is responsible for creating a

configuration data block that describes the resources available to the partition in which the instance is running. For example, the instance should not probe the underlying hardware to determine what resources are available for usage by the instance. Instead, if it is passed a configuration data block that describes what resources that instance is allowed to access, it will need to work with the specified resources.

[0277] 3. An instance may need to be capable of starting at an arbitrary physical address and may not be able to reserve any specific physical address in order to avoid conflicting with other operating systems running at that particular address.

[0278] 4. An instance may need to be capable of supporting multiple arbitrary physical holes in its address space, if it is part of a system configuration in which memory is shared between partitions. In addition, an instance may need to deal with physical holes in its address space in order to support "hot inswap" of memory.

[0279] 5. An instance may need to pass messages and receive notifications that new resources are available to partitions and instances. More particularly, a protocol is needed to inform an instance to search for a new resource. Otherwise, the instance may never realize that the resource has arrived and is ready for use.

[0280] 6. An instance may need to be capable of running entirely within its "private memory" if it is used in a system where instances do not share memory.

[0281] Alternatively, an instance may need to be capable of using physical "shared memory" for communicating or sharing data with other instances running within the computer if the instance is part of a system in which memory is shared. In such a shared memory system, an instance may need to be capable of mapping physical "shared memory" as identified in the configuration tree into its virtual address space, and the virtual address spaces of the "processes" running within that operating system instance.

[0282] 7. Each instance may need some mechanism to contact another CPU in the computer system in order to communicate with it.

[0283] 8. An instance may also need to be able to recognize other CPUs that are compatible with its operations, even if the CPUs are not currently assigned to its partition. For example, the instance may need to be able to ascertain CPU parameters, such as console revision number and clock speed, to determine whether it could run with that CPU, if the CPU was re-assigned to the partition in which the instance is running.

#### [0284] Changing the Configuration Tree

[0285] Each console program provides a number of callback functions to allow the associated operating system instance to change the configuration of the APMP system, for example, by creating a new community or partition, or altering the ownership of memory fragments. In addition, other callback functions provide the ability to remove a community, or partition, or to start operation on a newly-created partition.

[0286] However, callback functions do not cause any changes to take place on the running operating system instances. Any changes made to the configuration tree must be acted upon by each instance affected by the change. The type of action that must take place in an instance when the configuration tree is altered is a function of the type of change, and the operating system instance capabilities. For example, moving an input/output processor from one partition to another may require both partitions to reboot. Changing the memory allocation of fragments, on the other hand, might be handled by an operating system instance without the need for a reboot.

[0287] Configuration of an APMP system entails the creation of communities and partitions, and the assignment of unassigned components. When a component is moved from one partition to another, the current owner removes itself as owner of the resource and then indicates the new owner of the resource. The new owner can then use the resource. When an instance running in a partition releases a component, the instance must no longer access the component. This simple procedure eliminates the complex synchronization needed to allow blind stealing of a component from an instance, and possible race conditions in booting an instance during a reconfiguration.

[0288] Once initialized, configuration tree nodes will never be deleted or moved, that is, their handles will always be valid. Thus, hardware node addresses may be cached by software. Callback functions which purport to delete a partition or a community do not actually delete the associated node, or remove it from the tree, but instead flag the node as UNAVAILABLE, and clear the ownership fields of any hardware resource that was owned by the software component.

[0289] In order to synchronize changes to the configuration tree, the root node of the tree maintains two counters (transient\_level and current\_level). The transient\_level counter is incremented at the start of an update to the tree, and the current\_level counter is incremented when the update is complete. Software may use these counters to determine when a change has occurred, or is occurring to the tree. When an update is completed by a console, an interrupt can be generated to all CPUs in the APMP system. This interrupt can be used to cause system software to update its state based on changes to the tree.

#### [0290] Creation of an APMP Computer System

[0291] FIG. 5 is a flowchart that illustrates an overview of the formation of the illustrative adaptively-partitioned, multi-processor (APMP) computer system. The routine starts in step 500 and proceeds to step 502 where a master console program is started. If the APMP computer system is being created on power up, the CPU on which the master console runs is chosen by a predetermined mechanism, such as arbitration, or another hardware mechanism. If the APMP computer system is being created on hardware that is already running, a CPU in the first partition that tries to join the (non-existent) system runs the master console program, as discussed below.

[0292] Next, in step 504, the master console program probes the hardware and creates the configuration tree in step 506 as discussed above. If there is more than one partition in the APMP system on power up, each partition is initialized and its console program is started (step 508).

[0293] Finally, an operating system instance is booted in at least one of the partitions as indicated in step 510. The first operating system instance to boot creates an APMP database and fills in the entries as described below. APMP databases store information relating to the state of active operating system instances in the system. The routine then finishes in step 512. It should be noted that an instance is not required to participate in an APMP system. The instance can choose not to participate or to participate at a time that occurs well after boot. Those instances which do participate form a "sharing set." The first instance which decides to join a sharing set must create it. There can be multiple sharing sets operating on a single APMP system and each sharing set has its own APMP database.

#### [0294] Deciding to Create a New APMP System or to Join an Existing APMP System

[0295] An operating system instance running on a platform which is also running the APMP computer system does not necessarily have to be a member of the APMP computer system. The instance can attempt to become a member of the APMP system at any time after booting. This may occur either automatically at boot, or after an operator-command explicitly initiates joining. After the operating system is loaded at boot time, the operating system initialization routine is invoked and examines a stored parameter to see whether it specifies immediate joining and, if so, the system executes a joining routine which is part of the APMP computer system. An operator command would result in an execution of the same routine.

#### [0296] APMP Database

[0297] An important data structure supporting the inventive software allocation of resources is the APMP database which keeps track of operating system instances which are members of a sharing set. The first operating system instance attempting to set up the APMP computer system initializes an APMP database, thus creating, or instantiating, the inventive software resource allocations for the initial sharing set. Later instances wishing to become part of the sharing set join by registering in the APMP database associated with that sharing set. The APMP database is a shared data structure containing the centralized information required for the management of shared resources of the sharing set. An APMP database is also initialized when the APMP computer system is reformed in response to an unrecoverable error.

[0298] More specifically, each APMP database is a three-part structure. The first part is a fixed-size header portion including basic synchronization structures for creation of the APMP computer system, address-mapping information for the database and offsets to the service-specific segments that make up the second portion. The second portion is an array of data blocks with one block assigned to each potential instance. The data blocks are called "node blocks." The third portion is divided into segments used by each of the computer system sub-facilities. Each sub-facility is responsible for the content of, and synchronizing access to, its own segment.

[0299] The initial, header portion of an APMP database is the first part of the APMP database mapped by a joining operating system instance. Portions of the header are accessed before the instance has joined the sharing set, and, in fact, before the instance knows that the APMP computer system exists.

[0300] The header section contains:

- [0301] 1. a membership and creation synchronization quadword
- [0302] 2. a computer system software version
- [0303] 3. state information, creation time, incarnation count, etc.
- [0304] 4. a pointer (offset) to a membership mask
- [0305] 5. crashing instance, crash acknowledge bits, etc.
- [0306] 6. validation masks, including a bit for each service
- [0307] 7. memory mapping information (page frame number information) for the entire APMP database
- [0308] 8. offset/length pairs describing each of the service segments (lengths in bytes rounded to pages and offsets full pages) including
  - [0309] shared memory services
  - [0310] cpu communications services
  - [0311] membership services (if required)
  - [0312] locking services

[0313] The array of node blocks is indexed by a system partition id (one per instance possible on the current platform) and each block contains:

- [0314] instance software version
- [0315] interrupt reason mask
- [0316] instance state
- [0317] instance incarnation
- [0318] instance heartbeat
- [0319] instance membership timestamp
- [0320] little brother instance id and inactive-time; big brother instance id
- [0321] instance validation done bit.

[0322] An APMP database is stored in shared memory. The initial fixed portion of N physically contiguous pages occupies the first N pages of one of two memory ranges allocated by the first instance to join during initial partitioning of the hardware. The instance directs the console to store the starting physical addresses of these ranges in the configuration tree. The purpose of allocating two ranges is to permit failover in case of hardware memory failure. Memory management is responsible for mapping the physical memory into virtual address space for the APMP database.

[0323] The detailed actions taken by an operating system instance are illustrated in FIG. 6. More specifically, when an operating system instance wishes to become a member of a sharing set, it must be prepared to create the APMP computer system if it is the first instance attempting to "join" a non-existent system. In order for the instance to determine whether an APMP system already exists, the instance must be able to examine the state of shared memory as described above. Further, it must be able to synchronize with other instances which may be attempting to join the APMP system and the sharing set at the same time to prevent conflicting

creation attempts. The master console creates the configuration tree as discussed above. Subsequently, a region of memory is initialized by the first, or primary, operating system instance to boot, and this memory region can be used for an APMP database.

[0324] Mapping the APMP Database Header

[0325] The goal of the initial actions taken by all operating system instances is to map the header portion of the APMP database and initialize primitive inter-instance interrupt handling to lay the groundwork for a create or join decision. The routine used is illustrated in FIG. 6 which begins in step 600. The first action taken by each instance (step 602) is to engage memory management to map the initial segment of the APMP database as described above. At this time, the array of node blocks in the second database section is also mapped. Memory management maps the initial and second segments of the APMP database into the primary operating system address space and returns the start address and length. The instance then informs the console to store the location and size of the segments in the configuration tree.

[0326] Next, in step 604, the initial virtual address of the APMP database is used to allow the initialization routine to zero interrupt reason masks in the node block assigned to the current instance.

[0327] A zero initial value is then stored to the heartbeat field for the instance in the node block, and other node block fields. In some cases, the instance attempting to create a new APMP computer system was previously a member of an APMP system and did not withdraw from the APMP system. If this instance is rebooting before the other instances have removed it, then its bit will still be "on" in the system membership mask. Other unusual or error cases can also lead to "garbage" being stored in the system membership mask.

[0328] Next, in step 608, the virtual address (VA) of the APMP database is stored in a private cell which is examined by an inter-processor interrupt handler. The handler examines this cell to determine whether to test the per-instance interrupt reason mask in the APMP database header for work to do. If this cell is zero, the APMP database is not mapped and nothing further is done by the handler. As previously discussed, the entire APMP database, including this mask, is initialized so that the handler does nothing before the address is stored. In addition, a clock interrupt handler can examine the same private cell to determine whether to increment the instance-specific heartbeat field for this instance in the appropriate node block. If the private cell is zero, the interrupt handler does not increment the heartbeat field.

[0329] At this point, the routine is finished (step 610) and the APMP database header is accessible and the joining instance is able to examine the header and decide whether the APMP computer system does not exist and, therefore, the instance must create it, or whether the instance will be joining an already-existing APMP system.

[0330] Once the APMP header is mapped, the header is examined to determine whether an APMP computer system is up and functioning, and, if not, whether the current instance should initialize the APMP database and create the APMP computer system. The problem of joining an existing APMP system becomes more difficult, for example, if the

APMP computer system was created at one time, but now has no members, or if the APMP system is being reformed after an error. In this case, the state of the APMP database memory is not known in advance, and a simple memory test is not sufficient. An instance that is attempting to join a possibly existing APMP system must be able to determine whether an APMP system exists or not and, if it does not, the instance must be able to create a new APMP system without interference from other instances. This interference could arise from threads running either on the same instance or on another instance.

[0331] In order to prevent such interference, the create/join decision is made by first locking the APMP database and then examining the APMP header to determine whether there is a functioning APMP computer system. If there is a properly functioning APMP system, then the instance joins the system and releases the lock on the APMP database. Alternatively, if there is no APMP system, or if there is an APMP system, but it is non-functioning, then the instance creates a new APMP system, with itself as a member and releases the lock on the APMP database.

[0332] If there appears to be an APMP system in transition, then the instance waits until the APMP system is again operational or dead, and then proceeds as above. If a system cannot be created, then joining fails.

[0333] Creating a new APMP Computer System

[0334] Assuming that a new APMP system must be created, the creator instance is responsible for allocating the rest of the APMP database, initializing the header and invoking system services. Assuming the APMP database is locked as described above, the following steps are taken by the creator instance to initialize the APMP system (these steps are shown in FIGS. 7A and 7B):

[0335] Step 702 the creator instance sets the APMP system state and its node block state to "initializing."

[0336] Step 704 the creator instance calls a size routine for each system service with the address of its length field in the header.

[0337] Step 706 the resulting length fields are summed and the creator instance calls memory management to allocate space for the entire APMP database by creating a new mapping and deleting the old mapping.

[0338] Step 708 the creator instance fills in the offsets to the beginnings of each system service segment.

[0339] Step 710 the initialization routine for each service is called with the virtual addresses of the APMP database, the service segment and the segment length.

[0340] Step 712 the creator instance initializes a membership mask to make itself the sole member and increments an incarnation count. It then sets creation time, software version, and other creation parameters.

[0341] Step 714 the instance then sets itself as its own big and little brother (for heartbeat monitoring purposes as described below).

[0342] Step 716 the instance then fills in its instance state as "member" and the APMP system state as "operational."

[0343] Step 718 finally, the instance releases the APMP database lock.

[0344] The routine then ends in step 720.

[0345] Joining an Existing APMP Computer System

[0346] Assuming an instance has the APMP database locked, the following steps are taken by the instance to become a member of an existing APMP system (shown in FIGS. 8A and 8B):

[0347] Step 802 the instance checks to make sure that its instance name is unique. If another current member has the instance's proposed name, joining is aborted.

[0348] Step 804 the instance sets the APMP system state and its node block state to "instance joining"

[0349] Step 806 the instance calls a memory management routine to map the variable portion of the APMP database into its local address space.

[0350] Step 808 the instance calls system joining routines for each system service with the virtual addresses of the APMP database and its segment and its segment length.

[0351] Step 810 if all system service joining routines report success, then the instance joining routine continues. If any system service join routine fails, the instance joining process must start over and possibly create a new APMP computer system.

[0352] Step 812 assuming that success was achieved in step 810, the instance adds itself to the system membership mask.

[0353] Step 814 the instance selects a big brother to monitor its instance health as set forth below.

[0354] Step 816 the instance fills in its instance state as "member" and sets a local membership flag.

[0355] Step 818 the instance releases the configuration database lock. The routine then ends in step 820.

[0356] The loss of an instance, either through inactivity timeout or a crash, is detected by means of a "heartbeat" mechanism implemented in the APMP database. Instances will attempt to do minimal checking and cleanup and notify the rest of the APMP system during an instance crash. When this is not possible, system services will detect the disappearance of an instance via a software heartbeat mechanism. In particular, a "heartbeat" field is allocated in the APMP database for each active instance. This field is written to by the corresponding instance at time intervals that are less than a predetermined value, for example, every two milliseconds.

[0357] Any instance may examine the heartbeat field of any other instance to make a direct determination for some specific purpose. An instance reads the heartbeat field of another instance by reading its heartbeat field twice separated by a two millisecond time duration. If the heartbeat is not incremented between the two reads, the instance is considered inactive (gone, halted at control-P, or hung at or above clock interrupt priority level.) If the instance remains inactive for a predetermined time, then the instance is considered dead or disinterested.

[0358] In addition, a special arrangement is used to monitor all instances because it is not feasible for every instance

to watch every other instance, especially as the APMP system becomes large. This arrangement uses a "big brother-little brother" scheme. More particularly, when an instance joins the APMP system, before releasing the lock on the APMP database, it picks one of the current members to be its big brother and watch over the joining instance. The joining instance first assumes big brother duties for its chosen big brother's current little brother, and then assigns itself as the new little brother of the chosen instance. Conversely, when an instance exits the APMP computer system while still in operation so that it is able to perform exit processing, and while it is holding the lock on the APMP database, it assigns its big brother duties to its current big brother before it stops incrementing its heartbeat.

[0359] Every clock tick, after incrementing its own heartbeat, each instance reads its little brother's heartbeat and compares it to the value read at the last clock tick. If the new value is greater, or the little brother's ID has changed, the little brother is considered active. However, if the little brother ID and its heartbeat value are the same, the little brother is considered inactive, and the current instance begins watching its little brother's little brother as well. This accumulation of responsibility continues to a predetermined maximum and insures that the failure of one instance does not result in missing the failure of its little brother. If the little brother begins incrementing its heartbeat again, all additional responsibilities are dropped.

[0360] If a member instance is judged dead, or disinterested, and it has not notified the APMP computer system of its intent to shut down or crash, the instance is removed from the APMP system. This may be done, for example, by setting the "bugcheck" bit in the instance primitive interrupt mask and sending an IP interrupt to all CPU's of the instance. As a rule, shared memory may only be accessed below the hardware priority of the IP interrupt. This insures that if the CPUs in the instance should attempt to execute at a priority below that of the IP interrupt, the IP interrupt will occur first and thus the CPU will see the "bugcheck" bit before any lower priority threads can execute. This insures the operating system instance will crash and not touch shared resources such as memory which may have been reallocated for other purposes when the instances were judged dead. As an additional or alternative mechanism, a console callback (should one exist) can be invoked to remove the instance. In addition, in accordance with a preferred embodiment, whenever an instance disappears or drops out of the APMP computer system without warning, the remaining instances perform some sanity checks to determine whether they can continue. These checks include verifying that all pages in the APMP database are still accessible, i.e. that there was not a memory failure.

[0361] Assignment of Resources After Joining

[0362] A CPU can have at most one owner partition at any given time in the power-up life of an APMP system. However, the reflection of that ownership and the entity responsible for controlling it can change as a result of configuration and state transitions undergone by the resource itself, the partition it resides within, and the instance running in that partition.

[0363] CPU ownership is indicated in a number of ways, in a number of structures dictated by the entity that is managing the resource at the time. In the most basic case, the

CPU can be in an unassigned state, available to all partitions that reside in the same sharing set as the CPU. Eventually that CPU is assigned to a specific partition, which may or may not be running an operating system instance. In either case, the partition reflects its ownership to all other partitions through the configuration tree structure, and to all operating system instances that may run in that partition through the AVAILABLE bit in the HWRPB per-CPU flags field.

[0364] If the owning partition has no operating system instance running on it, its console is responsible for responding to, and initiating, transition events on the resources within it. The console decides if the resource is in a state that allows it to migrate to another partition or to revert back to the unassigned state.

[0365] If, however, there is an instance currently running in the partition, the console relinquishes responsibility for initiating resource transitions and is responsible for notifying the running primary of the instance when a configuration change has taken place. It is still the facilitator of the underlying hardware transition, but control of resource transitions is elevated one level up to the operating system instance. The transfer of responsibility takes place when the primary CPU executes its first instruction outside of console mode in a system boot.

[0366] Operating system instances can maintain ownership state information in any number of ways that promote the most efficient usage of the information internally. For example, a hierarchy of state bit vectors can be used which reflect the instance-specific information both internally and globally (to other members sharing an APMP database).

[0367] The internal representations are strictly for the use of the instance. They are built up at boot time from the underlying configuration tree and HWRPB information, but are maintained as strict software constructs for the life of the operating system instance. They represent the software view of the partition resources available to the instance, and may—through software rule sets—further restrict the configuration to a subset of that indicated by the physical constructs. Nevertheless, all resources in the partition are owned and managed by the instance—using the console mechanisms to direct state transitions—until that operating system invocation is no longer a viable entity. That state is indicated by halting the primary CPU once again back into console mode with no possibility of returning without a reboot.

[0368] Ownership of CPU resources never extends beyond the instance. The state information of each individual instance is duplicated in an APMP database for read-only decision-making purposes, but no other instance can force a state transition event for another's CPU resource. Each instance is responsible for understanding and controlling its own resource set; it may receive external requests for its resources, but only it can make the decision to allow the resources to be transferred.

[0369] When each such CPU becomes operational, it does not set its AVAILABLE bit in the per-CPU flags. When the AVAILABLE bit is not set, no instance will attempt to start, nor expect the CPU to join in SMP operation. Instead, the CPU, in console mode, polls the owner field in the configuration tree waiting for a valid partition to be assigned. Once a valid partition is assigned as the owner by the primary console, the CPU will begin operation in that partition.

[0370] During runtime, the current owner field reflects the partition where a CPU is executing. The AVAILABLE bit in the per-CPU flags field in the HWRPB remains the ultimate indicator of whether a CPU is actually available, or executing, for SMP operation with an operating system instance, and has the same meaning as in conventional SMP systems.

[0371] It should be noted that an instance need not be a member of a sharing set to participate in many of the reconfiguration features of an APMP computer system. An instance can transfer its resources to another instance in the APMP system so that an instance which is not a part of a sharing set can transfer a resource to an instance which is part of the sharing set. Similarly, the instance which is not a part of the sharing set can receive a resource from an instance which is part of the sharing set.

#### [0372] Shared Memory

[0373] Through software configuration, recorded in the console configuration tree, some memory is marked as shared among all instances in a community. Some memory is marked as private to a partition which can be running an instance of an operating system. All other memory is marked as unowned. Since the configuration is defined by software, it is possible to dynamically change partitions and the relative sizes of partitions.

[0374] All memory within the physical hardware system is associated with an owner field within the configuration tree. Memory can be owned by a partition, in which case the memory is used as private memory by the operating system or console software running within the partition. This is referred to as "private" memory. Alternatively, memory may be owned by a community, in which case, the memory is shared for all instances within the community and such memory is referred to as "shared" memory. Memory can also be configured to be owned by no partition or community. Such "unowned" memory may be powered down and out-swapped while the remainder of the system continues to operate, if hardware allows such "hot out-swapping".

[0375] Memory can be shared among instances in a community through the use of shared memory regions. A shared memory region can be created by any instance. A tag is specified to coordinate access to the same region by multiple instances. A virtual size is specified as well as a physical size. The virtual size may be the same size or larger than the physical size. Shared memory is initialized by a call-back routine. A lock is held during initialization to block out other instances from mapping to the region while the region is being initialized. Once the shared memory region is created by one instance, other instances can be mapped and attached to the region. A zero page table entry is used to indicate pages that are part of the region virtually but do not have physical memory associated with them. The memory region data structure records which instances have attached to the region. An instance must specify a call-back routine when attaching to a shared memory region. This routine is called for a variety of reasons: during initialization or shutdown of the system, or whenever another instance is attaching to or detaching from the region, or whenever an instance that was attached has crashed (detached in an unordered fashion). Depending upon the call-back reason, during shutdown, for example, the call-back routine is expected to block access to the shared memory region.

[0376] An instance can request that more physical memory be added to a region. Only the instance that makes

this request initially maps these new pages. When another instance attempts to access these pages, an access violation handler gains control (because that instance will have a zero page table entry (PTE) associated with that memory region), and the access violation handler updates the mapping region with any new pages. When an instance unmaps the region, the detachment is recorded in the APMP database. When all instances have detached from a region, it can be deleted and all pages released to a shared memory free page list. The shared memory region data structure also records which instances have outstanding I/O on any page within the region. The operating system instances record their individual reference counters such that they know when to set and clear their I/O bit in the region. An instance cannot unmap and detach from a shared memory region if it has outstanding I/O to any page within the region.

[0377] A shared memory API is a set of routines that can be called by user mode applications and maps shared memory into the application's address space. When a shared memory region is created, as described above, the associated creating instance keeps track of how the operating system's data structures relate to the shared memory region. When the instance has created data structures for the shared memory region, the instance is attached to the region. Then, when the shared memory mapping API routine is called, normal operating system mechanisms are used to map the application address space to the shared memory. When the local operating system data structures are cleaned up, the instance is detached from the region. A global section may be associated with a shared memory region in a one-to-one fashion.

[0378] An instance may specify a "context variable" which is to be associated with a region. If another instance attempts to attach to a region and does not specify the same context, an error is returned. This specification of a context variable may be used, for example, to associate a version number with the application. Additionally, an instance may specify a private context variable to be associated with the instance private data stored for a region. When the call-back routine is called, the instance can gather additional information about the region by obtaining the private context variable. The private context may be used, for example, to store a port number.

[0379] Shared memory can be borrowed by an operating system instance for use as instance private memory. Shared memory can be borrowed through the use of the shared memory API. Shared memory can be created, then used by only the local instance. This technique is useful if not all memory marked as shared is being used by the community member instances. The extra shared memory can be a pooled source of free memory. In other words, shared memory can be borrowed by the creation of a shared memory region. The pages in the shared memory region can be used by the local operating system for various purposes.

[0380] Private memory can be configured to be owned by the instance whose CPU(s) have fastest access to the memory. Nonuniform memory access is accommodated in the design's shared memory by organizing internal data structures for shared memory in groups according to the hardware characteristics of the memory. These internal data structures are called common property partitions. The shared memory API allows for memory characteristics to be speci-



fied by the caller. These characteristics can be expressed as nonuniform memory access properties such as "near" or "far".

[0381] The PFN database accommodates private memory and shared memory and reconfigured memory using a large array of page frame number (PFN) database entries. There is no physical memory behind a virtual array that describes pages that are private to another instance, nor corresponding to memory locations supported by memory boards that are missing from the system, nor corresponding to physical memory addressing holes. The layout of the PFN database suggests a particular granularity of physical memory. That is, in order to allocate and consume an integral number of physical pages for the PFN database that is to reside within each block of memory, physical memory should have a granularity as described below. The granularity of physical memory is chosen as the the smallest amount of memory that contains an integral number of pages and an integral number of PFN database entries. This is given by the least common multiple of the memory page size and the page frame number database entries, in quad words.

[0382] As described above, a creating instance, more specifically, the APMP computer system's initialization program, walks the configuration tree and builds management structures for its associated community's shared memory. In general, four hierarchical access modes provide memory access control. The access modes are, from the most to least privileged: kernel, executive, supervisor and user. Additionally, memory protection is specified at individual page level, where a page may be inaccessible, read only, or read/write for each of the four access modes. Accessible pages can be restricted to have only data or instruction access. Memory management software maintains tables of mapping information (page tables) that keep track of where each virtual page is located in physical memory. A process, through a memory management unit, utilizes this mapping information when it translates virtual addresses to physical addresses. The virtual address space is broken into units of relocation, sharing, and protection pages, which are referred to as pages. An operating system instance controls the virtual-to-physical mapping tables and saves the inactive parts of the virtual memory address space on external storage media.

[0383] Memory management employs, illustratively, a quad word page table entry to translate virtual addresses to physical addresses. Each page table entry (PTE) includes a page frame number (PFN) which points to a page boundary and may be concatenated with a byte-within-page indicator of a virtual address to yield a physical address.

[0384] Physical address translation is performed by accessing entries in a multi-level page structure. A page table base register (PTBR) contains the physical PFN of the highest level page table. Bits of the virtual address are used to index into the higher level page tables to obtain the physical PFNs of the base lower level page tables and, at the lowest level, to obtain the physical PFN of the page being referenced. This PFN is concatenated with the virtual address byte-within-page indicator to obtain the physical address of the location being accessed.

[0385] As noted above, an instance may decide to join the operation of a community at any time, not necessarily at system boot time. When an instance decides to join the APMP system, it calls a routine DB\_MAP\_initial, which

obtains the APMP data base pages from the configuration tree community node and maps the initial piece of the APMP database. If the configuration tree does not contain APMP database pages yet, the instance chooses shared memory pages to be used for the APMP database. The instance calls console code to write to the configuration tree in an asynchronous manner. After mapping the initial piece of the APMP database, it is determined as described above whether the instance is creating or joining the APMP system.

[0386] If the instance is the creator of the APMP system, the instance calls a routine, DB\_allocate, to allocate the pages for the APMP database and to initialize the mapping information within a MMAP data structure. The MMAP data structure, which is discussed in greater detail below, is used to describe a mapping of shared memory. The routine DB\_allocate does not unmap the initial piece of the APMP database. If the instance is a joiner of a APMP system, the instance calls a routine DB\_Map\_continue to map the APMP database. The routine DB\_Map\_continue does not unmap the initial piece of the APMP database. Once the APMP database is mapped and the joining instance's code has switched to referencing the newly mapped APMP database, rather than the initial APMP database, the initial APMP database is unmapped by calling a routine, DB\_unmap. This routine can also be called to unmap the APMP database when an instance is leaving the APMP system.

[0387] The APMP database need not be located at the same virtual location for all instances, as this would prevent instances from joining the APMP system if a given range of virtual addresses were unavailable. This flexibility permits different operating systems having different virtual address space layouts to readily coexist in the new APMP system.

[0388] The DB\_Map\_initial routine maps the initial piece of the APMP database, accepts the length of the initial APMP database and returns the virtual address of the initial APMP database. Additionally, DB\_Map\_initial will test the mapped pages to ensure that the pages are from shared memory and to mark any bad pages.

[0389] The DB\_allocate routine accepts the full address of the initial APMP database, the length of the initial APMP database, and the length of the entire APMP database. The routine returns the virtual address of the entire APMP database. The routine allocates sufficient instance address space to map the entire APMP database and remaps the initial piece of the APMP database in the beginning of this space. More APMP database pages are mapped from shared memory, as necessary. These pages may be tested and if a bad page is encountered, it is marked as used. The rest of the APMP database pages are mapped in the appropriate page table entries. Contiguous pages are allocated for the APMP database PFN list. The APMP database PFNs are stored in the PFN list pages, with any unused entries zeroed out. If enough contiguous pages are available for the entire APMP database, no PFN list pages are used. Shared pages are allocated directly from configuration tree and are taken from the page directly after the initial APMP database pages.

[0390] The DB\_Map\_continue routine maps the entire APMP database if a caller is not the creator of the APMP system. The routine accepts the virtual address of the initial APMP database and the length of the initial APMP database. The routine returns the starting virtual address of the entire APMP database and the length of the entire APMP database.

[0391] Each operating system instance includes memory configuration information functions which focus on the memory aspects of the configuration tree. A MEM-CONFIG\_INFO routine returns basic memory configuration information by reading the configuration tree fields MAX\_DESC and MAX\_FRAGMENTS and returning the maximum number of memory descriptor nodes and the maximum number of memory fragments per descriptor node. A MEM-CONFIG\_PFN routine determines which partition owns a given PFN. This routine accepts a page frame number and returns an indication of what type of page it is, that is, whether the page is shared or private to a particular partition, an input/output (I/O) page, or unowned memory. Additionally, if the page is private or used to access I/O devices, the routine returns an indication of which partition owns this PFN and, if the page is shared, which community owns the PFN. The SHMEM\_CONFIG\_DESC routine returns shared memory information about a memory descriptor in the configuration tree. Once a memory descriptor node is found, the routine searches the memory fragments for those fragments that are marked shared and fills in a return buffer with the PFN and page count for each fragment. If there are no memory fragments marked shared, a fragment count is set to zero.

[0392] The SHMEM\_CONFIG\_ALL routine returns information about all memory descriptor nodes that contain shared memory. The routine calls SHMEM\_CONFIG\_DESC in a loop to obtain all shared memory page ranges. Input arguments include the maximum number of memory descriptor nodes and the maximum number of memory fragments per descriptor node. The routine returns the total number of shared memory fragments from an array of structures that describe the shared memory ranges.

[0393] A routine SHMEM\_CONFIG\_APMP sets up the APMP PFN range in the configuration tree. If the APMP PFN range has already been set up, it returns the information. The routine returns the first PFN to use for the APMP database and number of APMP pages. This routine reads a value within a community node and if the value is zero, it obtains the first contiguous range of shared memory, eight megabytes in the illustrative embodiment, by calling the SHMEM\_CONFIG\_DESC routine. Then it calls the console dispatch routine to set this range in the community node. If there was a race to set the APMP page range, the range set in the configuration tree will be read and returned to the caller.

[0394] A shared memory management data structure in the APMP database SHMEM includes version number, the size of the fixed part of the SHMEM structure, flags that indicate whether the shared memory is valid, whether initialization is in progress, whether debug structure formats are being used, whether all pages within all shared memory common property partitions have been tested, and the maximum number of shared memory common property partitions. Additionally, the data structure includes the total number of valid shared memory common property partitions, the size of one shared memory common property partition structure, offsets from the beginning of the shared memory data structure to the shared memory common property partition array, an offset from the beginning of the shared memory data structure to the shared memory lock structure, a shared memory lock handle, and the maximum number of shared memory regions supported within the APMP system. The data struc-

ture also includes the total number of valid shared memory regions and an offset from the beginning of the shared memory data structure to the shared memory region tag array. The size of a shared memory region structure, and the offset from the beginning of the shared memory management data structure to the shared memory region array is also included.

[0395] Instance private memory data cells contain information about the shared memory management area in the APMP database. This information includes a pointer to the beginning of the shared memory data structure and the same descriptors as were described in relation to the shared memory data structure: the maximum number of shared memory common property partitions, maximum number of memory fragments in each shared memory common property partition, the size of one shared memory common property partition structure, a pointer to a shared memory common property partition array within the APMP database, a pointer to a shared memory list and a pointer to a shared memory region tag array within the APMP database. Additionally, the maximum number of shared memory regions, the size of one shared memory region structure, a pointer to a shared memory region array within the APMP database, and a pointer to the shared memory descriptor array in private memory are included.

[0396] When a shared memory common property partition (CPP) configuration area is initialized, the APMP database pages are excluded. Shared memory common property partitions support hot-swapping and non-uniform memory access by partitioning shared memory into partitions having common properties. Flags and routines are employed to indicate, for example, which non-uniform memory access unit a CPP is in, or which hot swappable unit a CPP is in, along with the range and location of memory pages within the unit. Each instance that is a member of an APMP system maintains data within its own private memory regarding each shared memory CPP that it is connected to. A lock structure is employed to synchronize access to the shared memory common property partition data structure. The lock is held when a partition is connecting to the shared memory CPP, when a partition is disconnecting from a shared memory CPP, when pages are being allocated from the shared memory CPP, or when pages are being deallocated to the shared memory CPP. Each shared memory CPP has a free page list, a bad page list, and an untested page list. Pages can be allocated from the free page and untested page lists and deallocated into the free page list and bad page list. The shared memory CPP page list links are maintained within the PFN database entries for the pages.

[0397] The shared memory lock is employed to synchronize the SHM\_TAG array and an associated list of valid SHM\_REG structures and to synchronize access to the list of free SHM\_REG structures. The SHMEM lock must be held while reading or writing the SHM\_TAG array, while manipulating a list of valid SHM\_REG structures, or while manipulating the free SHM\_REG list. Shared memory locks are ranked as follows: the highest order lock is an IPL 8 SMP spinlock, followed by the SHM\_CPP lock, the SHM\_REG lock and, finally, the SHMEM lock. For example, while holding the SHMEM lock, one can acquire a SHM\_REG lock, a SHM\_CPP lock and/or a SMP spinlock, in that order. Shared memory management functions can be called from

kernel mode to get information about shared memory. The SHMEM lock has a ranking relative to other locks such that no deadlocks occur.

[0398] A software implementation of the above-described embodiment may comprise a series of computer instructions either fixed on a tangible medium, such as a computer readable media, e.g. diskette 142, CD-ROM 147, ROM 115, or fixed disk 152 of FIG. 1, or transmittable to a computer system, via a modem or other interface device, such as communications adapter 190 connected to the network 195 over a medium 191. Medium 191 can be either a tangible medium, including but not limited to optical or analog communications lines, or may be implemented with wireless techniques, including but not limited to microwave, infrared or other transmission techniques. It may also be the Internet. The series of computer instructions embodies all or part of the functionality previously described herein with respect to the invention. Those skilled in the art will appreciate that such computer instructions can be written in a number of programming languages for use with many computer architectures or operating systems. Further, such instructions may be stored using any memory technology, present or future, including, but not limited to, semiconductor, magnetic, optical or other memory devices, or transmitted using any communications technology, present or future, including but not limited to optical, infrared, microwave, or other transmission technologies. It is contemplated that such a computer program product may be distributed as a removable media with accompanying printed or electronic documentation, e.g., shrink wrapped software, pre-loaded with a computer system, e.g., on system ROM or fixed disk, or distributed from a server or electronic bulletin board over a network, e.g., the Internet or World Wide Web.

[0399] Although an exemplary embodiment of the invention has been disclosed, it will be apparent to those skilled in the art that various changes and modifications can be made which will achieve some of the advantages of the invention without departing from the spirit and scope of the invention. For example, it will be obvious to those reasonably skilled in the art that, although the description was directed to a particular hardware system and operating system, other hardware and operating system software could be used in the same manner as that described. Other aspects, such as the specific instructions utilized to achieve a particular function, as well as other modifications to the inventive concept are intended to be covered by the appended claims.

What is claimed is:

1. A computer system having a plurality of system resources including processors, memory and I/O circuitry, the computer system comprising:

an interconnection mechanism for electrically interconnecting the processors, memory and I/O circuitry so that each processor has electrical access to all of the memory and at least some of the I/O circuitry;

a software mechanism for dividing the system resources into a plurality of partitions; and

at least one group of partitions forming a community which shares memory.

2. A computer system according to claim 1 wherein at least some of the memory is exclusively assigned to one of the partitions.

3. A computer system according to claim 1 wherein a segment of the shared memory forms a database for the community's shared memory.

4. A computer system according to claim 3 wherein the shared memory database indicates whether an operating system instance associated with a partition within the community is active.

5. A computer system according to claim 4 wherein each instance within a community maps to the shared memory database.

6. A computer system according to claim 5 wherein each instance examines the shared memory database as it maps to the database to determine whether other instances are active and, if other instances are active, joins the community.

7. A computer system according to claim 5 wherein each instance examines the shared memory database as it maps to the database to determine whether other instances are active and, if no other instances are active, creates the community.

8. A computer system according to claim 7 wherein an instance configures the shared memory database as it creates the community.

9. A computer system according to claim 6 wherein access to the shared memory database region is synchronized through use of locks which block access to the database by one instance while another instance is accessing the region.

10. A computer system according to claim 9 wherein each partition comprises a console program which controls the processors in a partition and wherein the computer system further comprises a mechanism for sending configuration information from a master console program to each of the other console programs.

11. A method for constructing a computer system having a plurality of system resources including processors, memory and I/O circuitry, the method comprising the steps of:

(a) electrically interconnecting the processors, memory and I/O circuitry so that each processor has electrical access to all of the memory and at least some of the I/O circuitry;

(b) dividing the system resources into a plurality of partitions; and

(c) forming a community of partitions by sharing memory among a group of partitions.

12. A method according to claim 11 wherein step (b) comprises the step of:

(b1) assigning at least some of the memory to each of the partitions.

13. A method according to claim 11 wherein step (c) comprises the step of:

(c1) forming a shared memory database for the community within the shared memory.

14. A method according to claim 13 wherein step (c1) comprises the step of:

(c1a) including within the shared memory database an indication of whether an operating system instance associated with the community is active.

15. A method according to claim 13 further comprising the step of:

- (d) each instance mapping to the shared memory database as it joins a community.
16. A method according to claim 14 further comprising the step of:
- (e) each instance examining the shared memory database to determine whether other instances are active and joining the community if other instances are active.
17. A method according to claim 16 wherein step (e) comprises the step of:
- (e1) creating the community if no other instance is active.
18. A method according to claim 17 wherein step (e) comprises the step of:
- (e2) reconfiguring the shared memory database.
19. A method according to claim 16 wherein step (e) comprises the step of:
- (e3) locking the shared memory database to block access to the database by another instance as an instance examines the shared memory database.
20. A computer program product for constructing a computer system having a plurality of system resources including processors, memory and I/O circuitry and hardware for electrically interconnecting the processors, memory and I/O circuitry so that each processor has electrical access to all of the memory and at least some of the I/O circuitry, the computer program product comprising a computer usable medium having computer readable program code thereon including:
- (a) program code for dividing the system resources into a plurality of partitions; and
- (b) program code for forming a community of partitions which shares memory.
21. A computer program product according to claim 20 wherein the program code for dividing the system resources into a plurality of partitions assigns at least some of the memory exclusively to one of the partitions.
22. A computer program product according to claim 21 wherein the program code for dividing the system resources into a plurality of partitions comprises program code for forming a database for the community's shared memory.
23. A computer program product according to claim 22 wherein the program code for forming a database for the community's shared memory comprises program code for indicating whether an operating system instances associated with a partition within the community is active.
24. A computer program product according to claim 23 wherein the computer program product for constructing a computer system having a plurality of system resources further comprises program code for mapping each operating system instance within a community to the shared memory database.
25. A computer program product according to claim 26 further comprising program code for each instance examining the shared memory database as the instance maps to the database, determining whether other instances are active, and joining the community if other instances are active, or creating the community if no other instances are active.
- \* \* \* \* \*